Software Architecture of Microprocessors and Microcontrollers

Webpage and Link to Moodle Course CA-SWB4-TIB4:
www.hs-esslingen.de/
mitarbeiter/Werner-Zimmermann
Aktuelle Vorlesungen – Computerarchitektur

Prof. Dr. rer. nat (Purdue Univ.) Jörg Friedrich
Prof. Dr.-Ing. Werner Zimmermann

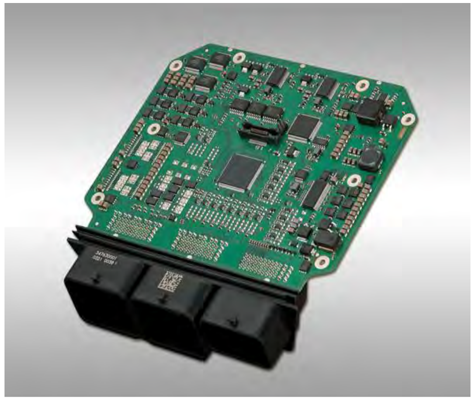
Hochschule Esslingen – University of Applied Sciences – Department of Information Technology

This is how a computer looks like …

… or like this?

This “gas” station is based on a computer too!

And some computers even make it into the news …



IT World Champions …

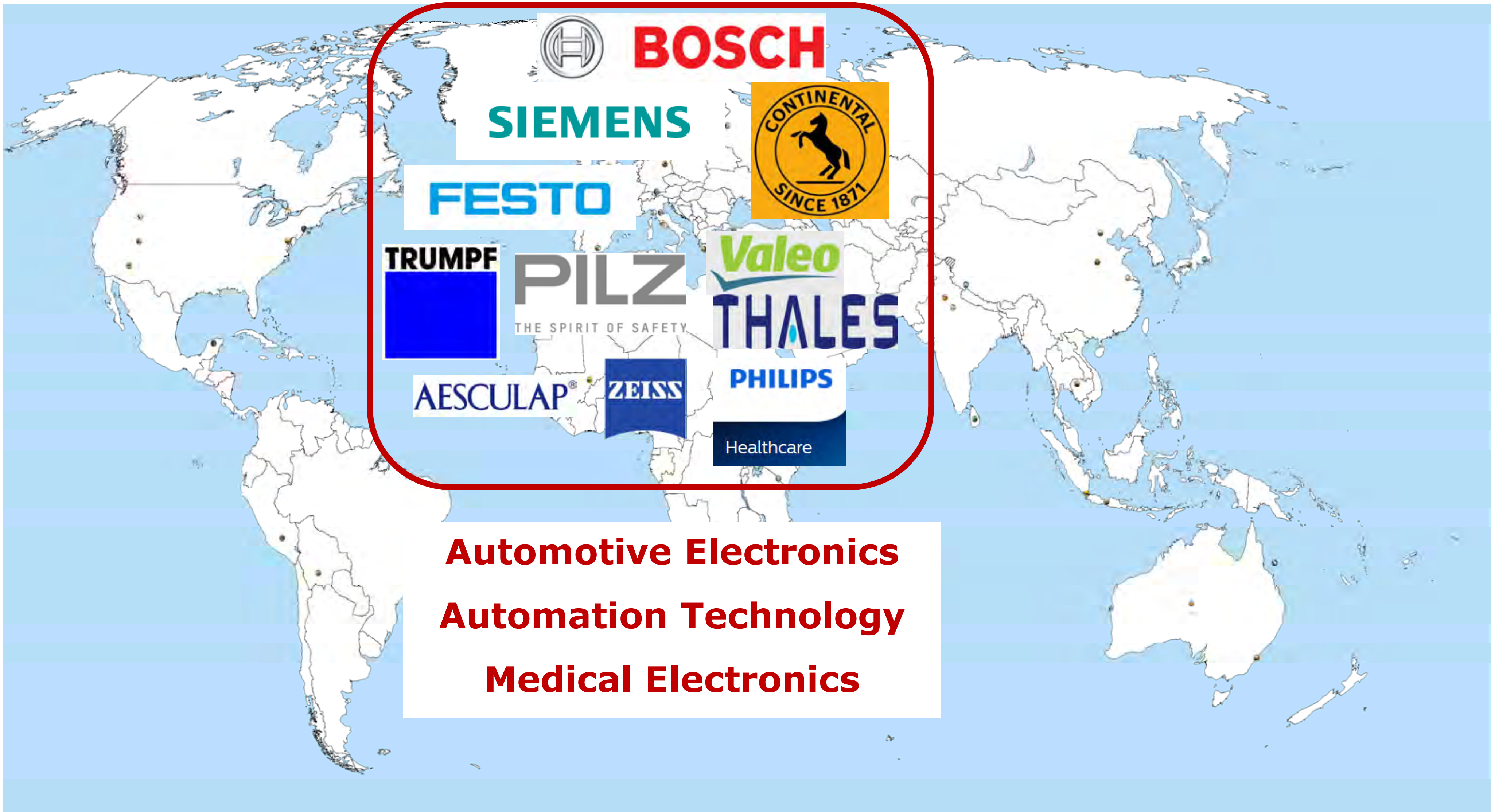
IBM
Microsoft
Apple
intel
Google
facebook
ORACLE
SAP
acer
htc
SAMSUNG
LG
SONY

Classical IT

Consumer Electronics Hardware

Something missing here?

... and finally Europe



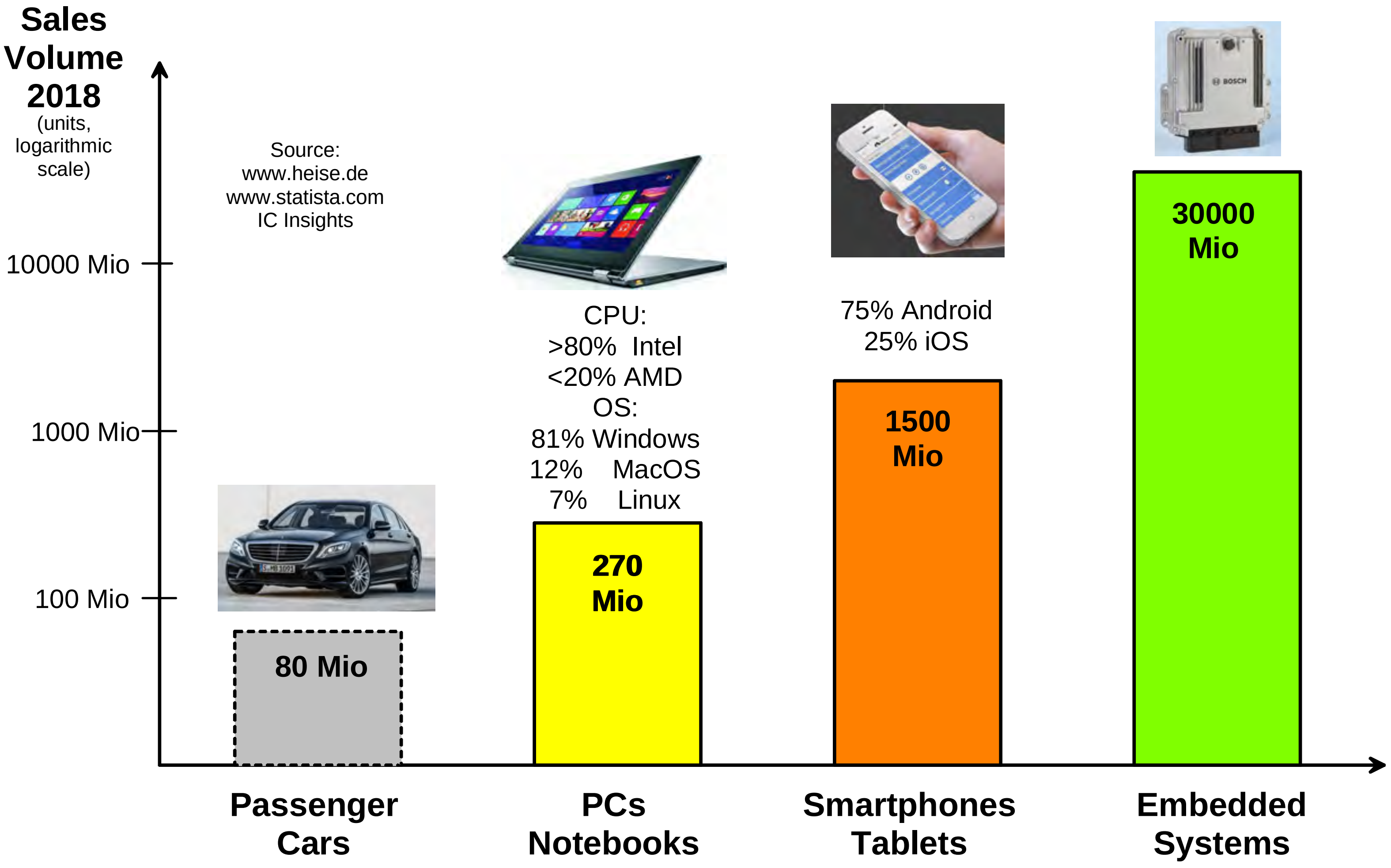
Automotive Electronics

Automation Technology

Medical Electronics

The Embedded World: Automotive, Industrial & Home Automation, Medical Systems

How important is it?



Source:
www.heise.de
www.statista.com
IC Insights

CPU:
>80% Intel
<20% AMD
OS:
81% Windows
12% MacOS
7% Linux

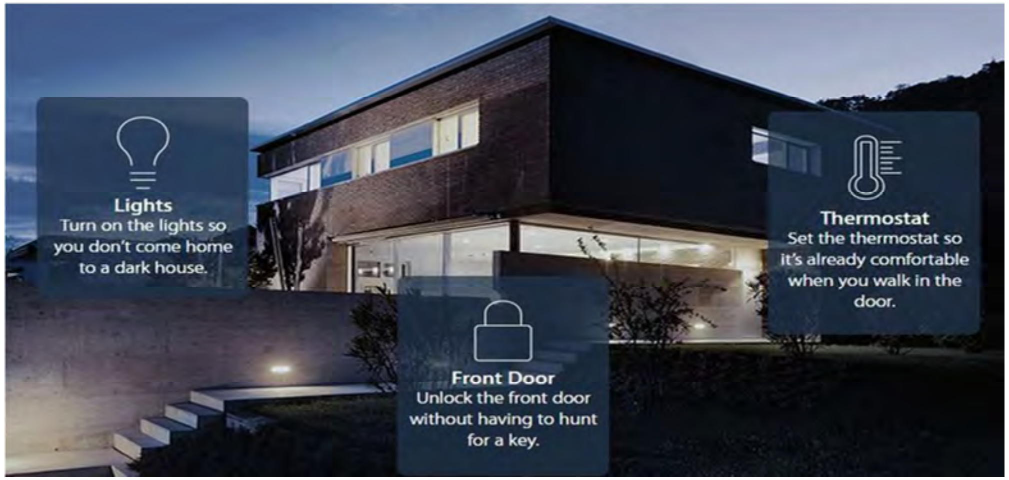
75% Android
25% iOS

… more to come: Internet of Things IoT

Everything will be a computer on a global network

Wearables

Home Automation



Lights
Turn on the lights so you don't come home to a dark house.

Thermostat
Set the thermostat so it's already comfortable when you walk in the door.

Front Door
Unlock the front door without having to hunt for a key.

Automated Cars

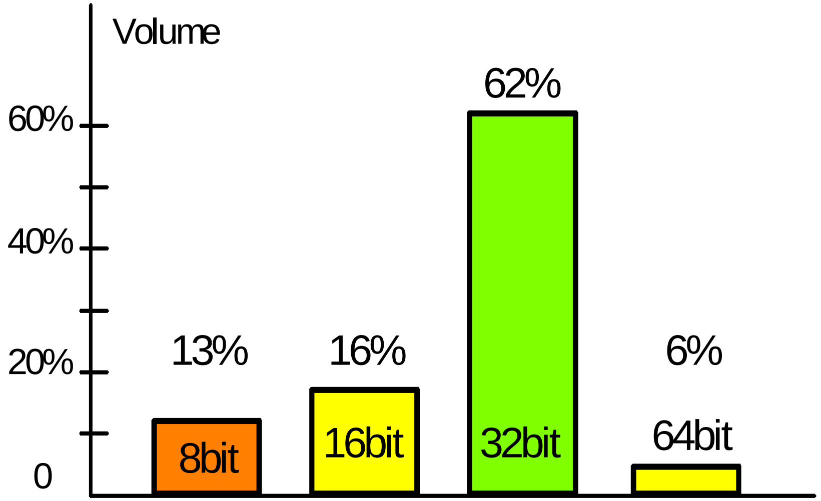
Mobility Solutions

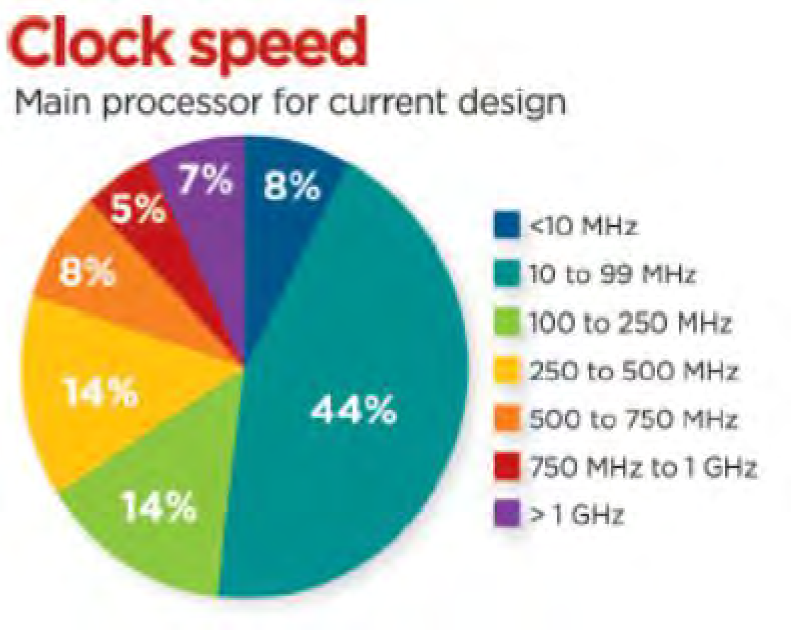
What is special with embedded systems?

- **Reliability and safety** a major issue → buggy software can kill people
- **Real time** requirements → computer must run in sync with external events
- **Resource constraints** → computing speed and memory size matter (cost!)

Embedded Design
Microprocessor market shares

(Source: EDN, 2015)





- Smaller bit size

Your washing machine does no 64bit processing!

- Moderate clock speed

It is not the CPU’s job to heat your toaster!

Sources: www.embedded.com, ESD Market Survey; Michael Barr, Real men program in C, www.embedded.com. Statista.com

Most popular Programming Languages

	Typical Application	TIOBE Rating	IEEE Rank
C	General Purpose	16%	3
C++	General Purpose	7%	4
Java	Business Applications, Android Apps	13,5%	2
Python	Numerical Analysis, AI, Testing	10,5%	1
C#	General Purpose	4,5%	7
Visual Basic	Business Application	4%	-
Javascript	Web Clients and Servers	2,5%	6
PHP	Web Servers	2,5%	-
SQL	Database	2%	-
Objective C+Swift	Apple IOS Apps	2%	9
R	Numerical Analysis	2%	5
Matlab	Numerical Analysis	1%	8
Assembler	Performance Sensitive Applications	1%	-
Ruby, Rust, Go, …		je < 1%	-
. . .	. . .	. . .	. . .
		Total 100%	Top 10

Sources (as of Sept 2020)

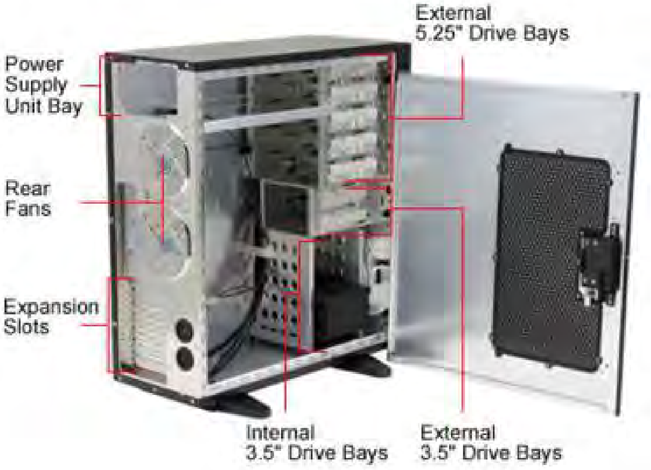
a) https://www.tiobe.com/tiobe-index/

b) https://spectrum.ieee.org/computing/software/the-top-programming-languages-2019

c) https://www.ahl.com/ahl-tech-the-curious-case-of-the-longevity-of-c

Large or small …

Main Frame



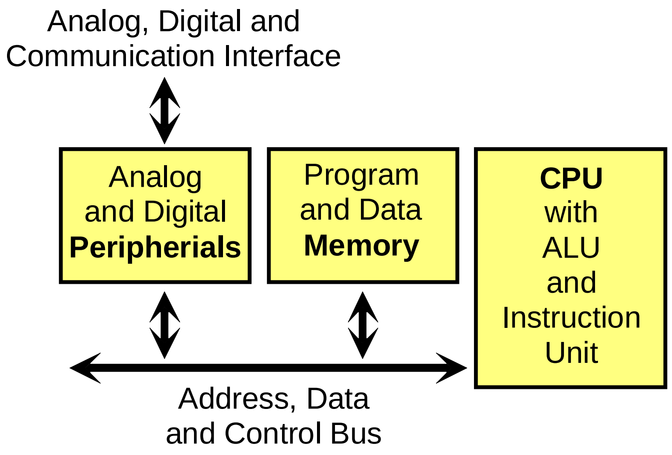
PC

Smartphone

Automotive Electronic Control Unit

… all computers have the same basic structure

Hard-ware



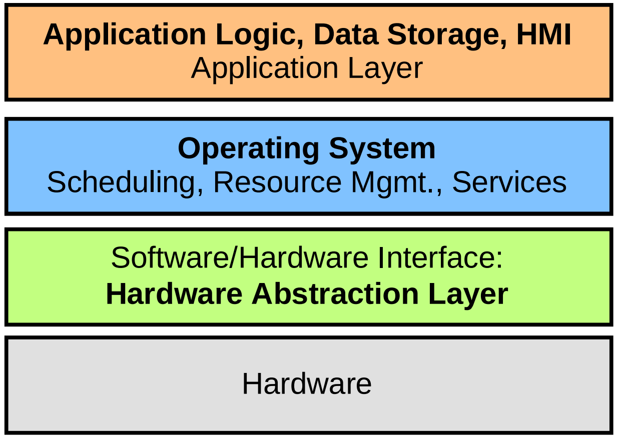
Analog, Digital and Communication Interface

Analog and Digital **Peripherials**

Program and Data **Memory**

CPU with ALU and Instruction Unit

Address, Data and Control Bus



Application Logic, Data Storage, HMI
Application Layer

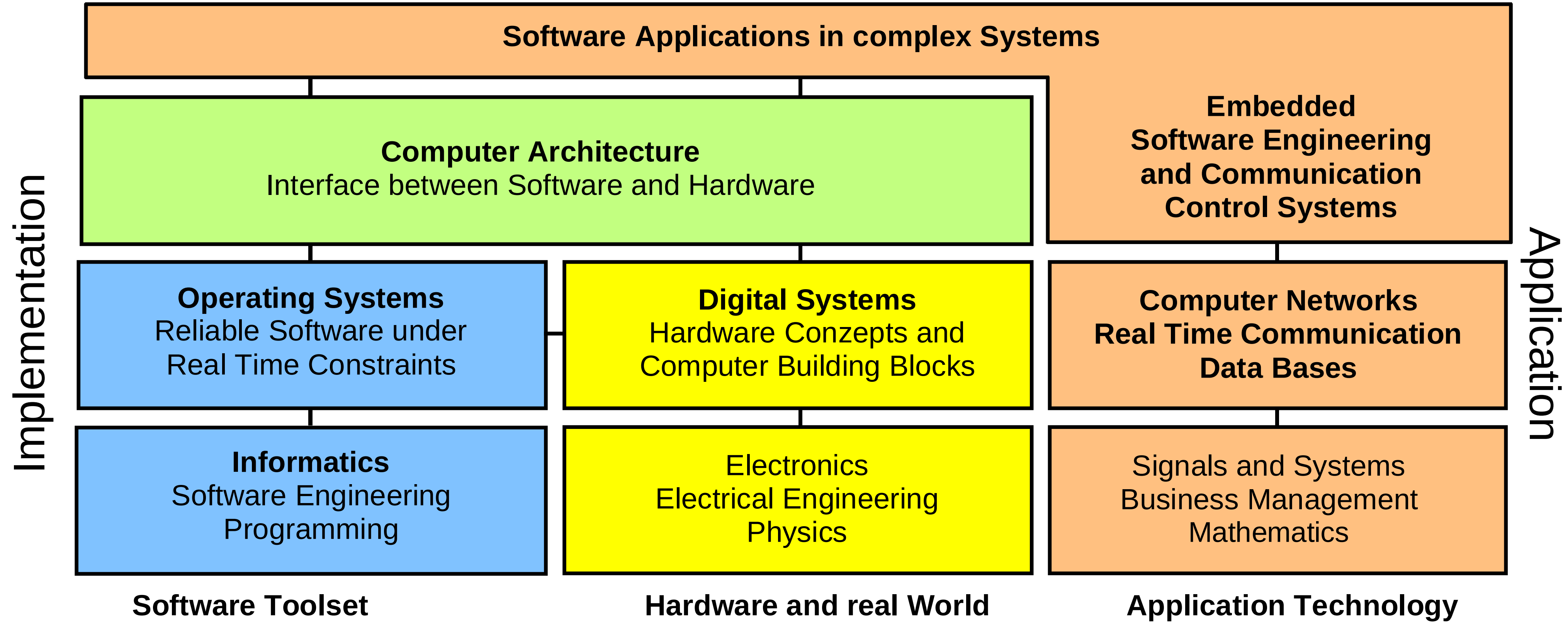
Operating System
Scheduling, Resource Mgmt., Services

Software/Hardware Interface:
Hardware Abstraction Layer

Hardware

Soft-ware

What do you need to know: Our IT curriculum



Software Applications in complex Systems

Computer Architecture
Interface between Software and Hardware

Embedded Software Engineering and Communication Control Systems

Operating Systems
Reliable Software under Real Time Constraints

Digital Systems
Hardware Conzepts and Computer Building Blocks

Computer Networks Real Time Communication Data Bases

Informatics
Software Engineering
Programming

Electronics
Electrical Engineering
Physics

Signals and Systems
Business Management
Mathematics

Literature

The following list is a small selection out of a large range of books about computer architecture, microprocessors and microcontrollers:

Introductory Books

Computer Architecture and Organization

[1.1] Patterson, D.; Hennessy, J.: Computer Organization and Design. Hardware-/Software-Interface. Morgan Kaufmann Publishers (Deutsche Übersetzung: Rechnerorganisation und –entwurf. Spektrum Akademischer Verlag)

[1.2] Hennessy, J.; Patterson, D.: Computer Architecture. A Quantitative Approach. Academic Press (Advanced topics from the same authors as [1.1])

[1.3] Tanenbaum, A.: Structured Computer Organization. Prentice Hall (Deutsche Übersetzung: Computerarchitektur. Pearson)

[1.4] Keller, R.; Lindermeir, W.; Marchthaler, R. Zimmermann, W. Digitaltechnik 1 + 2. Vorlesungsskript. Hochschule Esslingen

[1.5] Friedrich, J.: Echtzeitsysteme – Vorlesungsskript. Hochschule Esslingen

[1.6] Lewis, D.: Fundamentals of Embedded Software – Where C and Assembly Meet. Prentice Hall Verlag

[1.7] Beierlein, Hagenbruch: Taschenbuch Mikroprozessortechnik. Fachbuchverlag Leipzig

Microcontrollers

HCS12 Microcontroller Hardware and Software

[2.1] Friedrich, J.: Computerarchitektur 3 - Vorlesungsskript. Hochschule Esslingen

[2.2] Kreidl, H.; Kupris, G.; Thamm, O.: Mikrocontroller-Design. Hardware- und Softwareentwicklung mit dem 68HC12/HCS12. Hanser Verlag

[2.3] Huang, H.W.: The HCS12/9S12. An Introduction to Hardware and Software Interfacing. Thomson Learning

Literature

[2.4] Barret, S.; Pack, D.: Embedded Systems: Design and Applications with the 68HC12 and HCS12. Prentice Hall

[2.5] Cady, F.: Software and Hardware Engineering: Assembly and C-Programming for the Freescale HCS12 Microcontroller. Oxford University Press

[2.6] Almy, T.: Designing with Microcontrollers: The 68HCS12. Buchtext auf CD. http://www.hcs12text.com

Data Books

Freescale HCS12 Datasheets from
http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MC9S12DP256B#Data_Sheets

[3.0] Freescale Microcontroller MC9S12DP256 All-in-one-manual (outdated): 000-MC9S12DP256.pdf

[3.1] Freescale CPU-Architecture, Instruction Set, Operand Addressing: 001-S12CPUV2-ReferenceManual.pdf

[3.2] Freescale Microcontroller MC9S12DP256B Peripherals Overview: 002-9S12DP256BDGV2-DevicesUserGuide.pdf

[3.3] Freescale Digital-I/O, Interrupt System: 003-S12DP256PIMV2-Port Integration Module.pdf

[3.4] Freescale Clock and Reset Generator, Real Time Interrupt: 004-S12CRGV2-Clock&Reset-Generator.pdf

[3.5] Freescale Timer and Capture/Compare Inputs/Outputs: 005-S12-ECT_16B8CV1-Enhanced Capture Timer.pdf

[3.6] Freescale Serial Interface: 006-S1-2SCIV2-Serial Communication Interface.pdf

[3.7] Freescale PWM-Outputs: 007-S12PWM_8B8CV1-PWM.pdf

[3.8] Freescale Analog-Digital-Converter: 008-S12ATD10B8CV2-AnalogToDigital.pdf

Literature

[3.9] Freescale — CAN-Interface:
009-S12MSCANV2-CAN.pdf

[3.10] Freescale — Port A, B, E, K und Multiplex-Address-Data Bus
010-S12MEBIV3.pdf

[3.11] Wytec/EVBplus — Dragon12 MC9S12DP256 Development Board. Getting Started Manual and Circuit Diagrams. www.evbplus.com

[3.12] Freescale/Metroworks — Codewarrior Development Studio IDE User’s Guide.
8_16bit_IDE_Users_Guide.pdf

[3.13] Freescale/Metroworks — HC12/S12 Compiler. Compiler_HC12.pdf

[3.14] Freescale/Metroworks — HC12/S12 Assembler Manual. Assembler_HC12.pdf

[3.15] Freescale/Metroworks — Codewarrior Debugger. Debugger_HC12.pdf

[3.16] Freescale/Metroworks — HC12/S12 True Time Simulator. See [3.15]

[3.17] Freescale/Metroworks — Linker. Build_Tools_Utilities.pdf

Acknowledgements

The HCS12 microcontroller and the Dragon12 evaluation board were introduced by Prof. Dr. J. Friedrich (see [2.1]). Good job, thanks.

W. Zimmermann

Chapter 2
Architecture of a typical Microcontroller: Freescale HCS12

2.1 Basic Features of Microcontroller Family HCS12

- **Von Neumann** architecture — no caches, no MMU
- Complex Instruction Set (**CISC**) — many instructions and addressing modes
- Data word size n_{DAT}=**16 bit** → 16 bit CPU
- Address word size n_{ADR}=16 bit
- Smallest addressable unit n_{min}=1 Byte

→ Address space $N=2^{n_{ADR}} \cdot n_{min} = 2^{16}$ Byte = 64 KB

Extensible via memory banking (pages)

- No memory alignment, i.e. instructions and data can start at any address
- Multi-byte values stored in **Big Endian** (Most Significant Byte first) sequence
  Memory access requires the address of the first byte and the length of the data

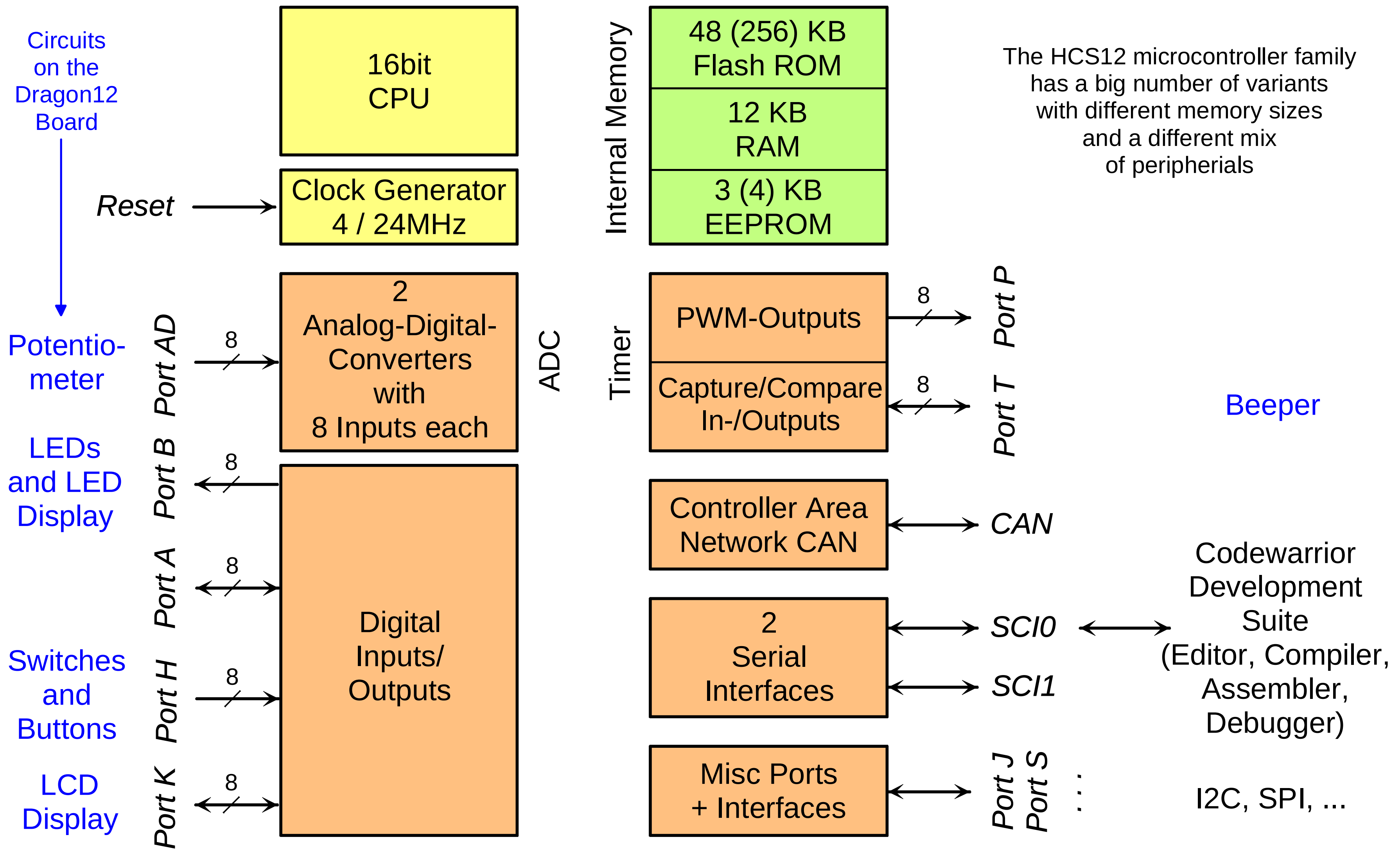
Example: 16bit value $4433 at address $0103

Address	*Content*	
0	. . .	
. . .	. . .	
$0103	$44	MSByte
$0104	$33	LSByte
. . .	. . .	

Some microprocessors, e.g. Intel x86, use Little Endian:
Least significant byte first

[*1] Motorola/Freescale use $. . . instead of . . . h or 0x. . . to mark hexadecimal values, e.g. $4433 = 4433h

Block Diagram of Microcontroller MC9S12DP256 (see [3.0 Page 16], [3.2 Fig. 1-1])



The HCS12 microcontroller family has a big number of variants with different memory sizes and a different mix of peripherials

Dragon12 Evaluation Board (see [3.11] and [2.1 Appendix B])

LCD Display
2 colums with 16 characters

LED 7-Segment-Display
4 digits

Freescale (Motorola)
MC9S12DP256 CPU

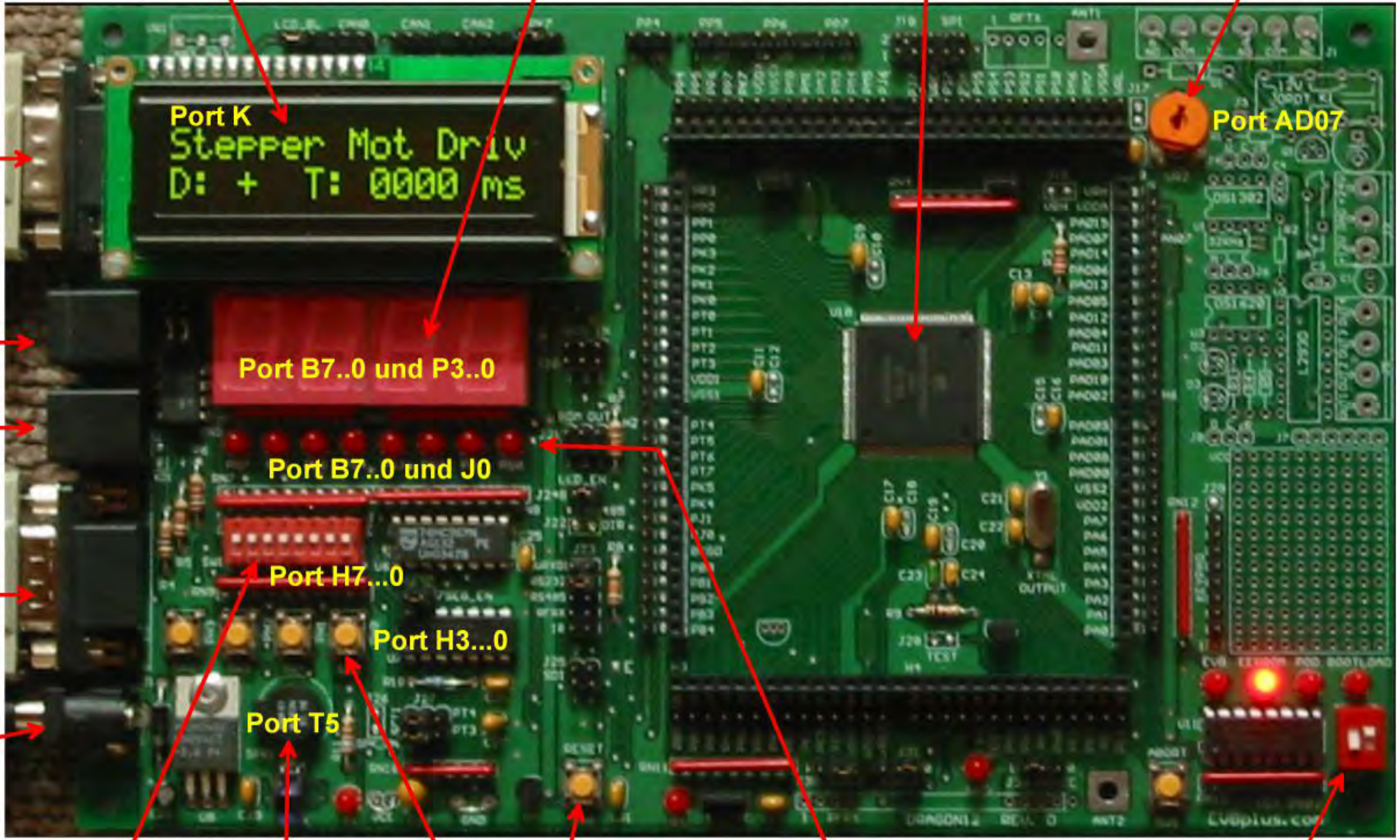
Poten-
tiometer

Serial Inter-face SCI0 (De-bugger)

CAN Inter-face

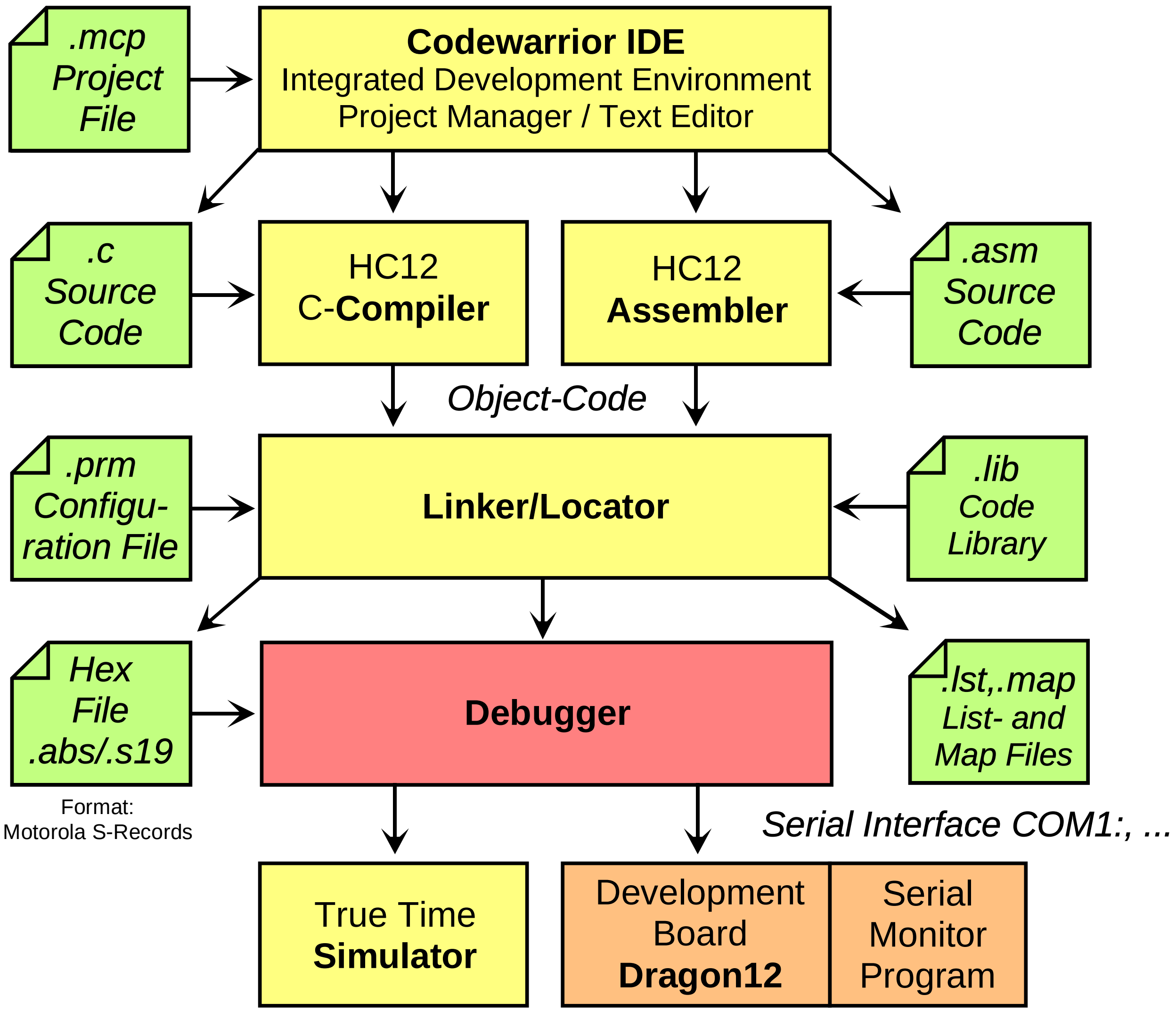
Serial Inter-face SCI1 (User Program)

Power Supply



8 DIP-Switches Beeper 4 Buttos Reset-Button 8 Single LEDs Debug Mode

Integrated Development Environment (IDE) Freescale CodeWarrior (see [3.12 – 3.17])



Format:
Motorola S-Records

Historical Note:

Semiconductor manufacturer Freescale in former times was part of Motorola.

The CodeWarrior tools were developed by Metroworks, which was purchased by Freescale.

MC9S12DP256 Memory Map (see [3.0 Page 120], [3.2 Fig. 1-2])

The HCS has several operating modes. In the lab we use the Normal Single Chip Mode without external memory.

$0000	HW/SW Interface: Registers to control the On-Chip-Peripherals 1KB	All peripherals are Memory-Mapped, i.e. for software their registers look like variables
$0400	EEPROM 3KB	The EEPROM has 4KB, but 1KB is shadowed by the peripheral registers.
$1000	RAM 12KB	Program variables Stack for debug monitor program at the end of the RAM area (36B)
$4000	Flash-ROM 16KB	Program code
$8000	Flash-ROM 16KB	This address range can be used to map additional 16KB Flash-ROM pages (Page Window selected by PPAGE-register) → Memory extension to > 64KB
$C000	Flash-ROM 16KB	$F780 … $FE00: Debugger Monitor Program $FF00 … $FFFF: Interrupt Vector Table 256B
$FFFF		

2.2 Hello Embedded World

Ever since Kernighan and Ritchie in their book „The C Programming Language", the first program to introduce a programing language used to be „**Hello World**", which outputs a short text string to the display. However, typical embedded systems don't have a keyboard and a display. So the idea is modified and digital output pins are toggled (**Toggle Port**), to which LEDs are connected (**Blinking LEDs**).

Converting the idea into a running program on the Dragon12 board takes some steps:

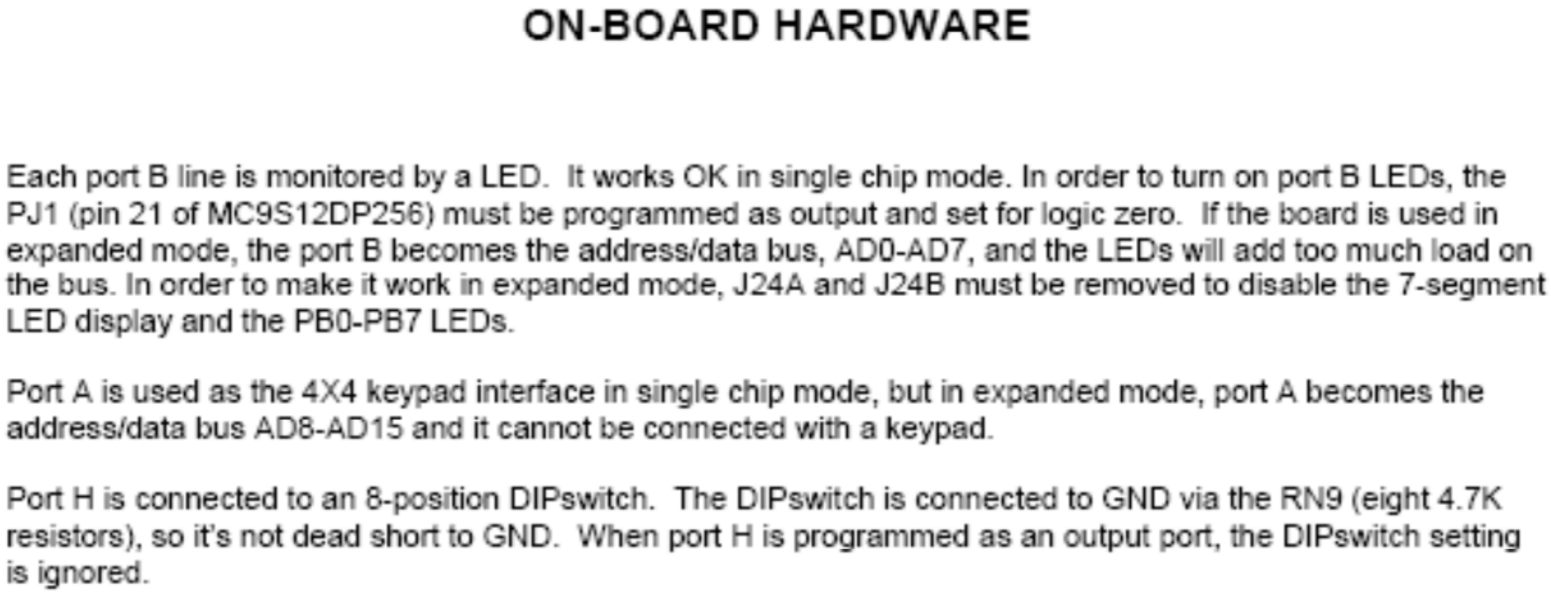
Step 1: What is the hardware setup of the evaluation board?
Where are the LEDs connected and how can they be controlled?

Info source [3.11]:

Getting_started_Dragon12.pdf S.11

Port J.1 Output
J.1= 0 Enable LED

Port B.7 … 0 Outputs
B.x=1 → LED on



ON-BOARD HARDWARE

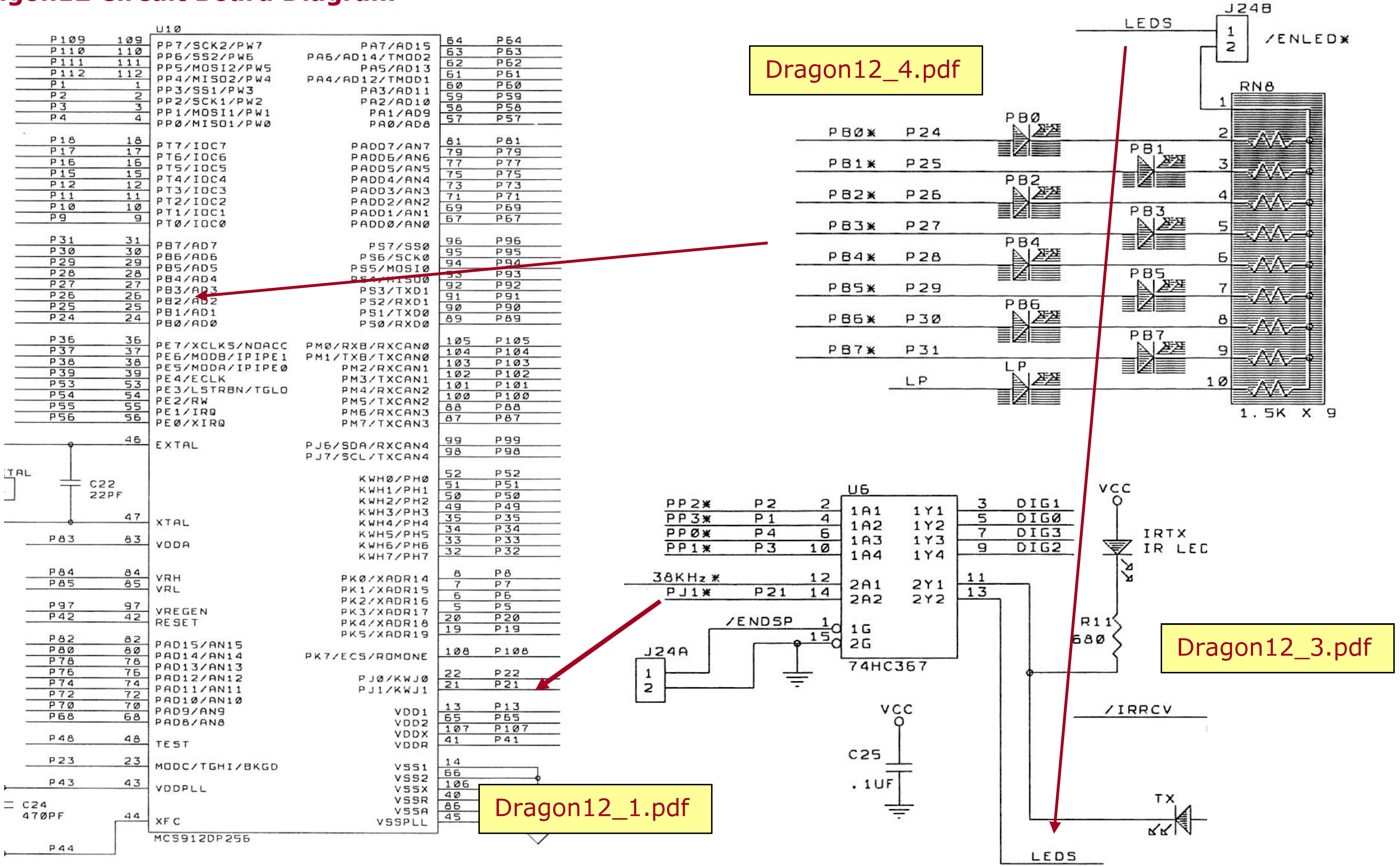
Each port B line is monitored by a LED. It works OK in single chip mode. In order to turn on port B LEDs, the PJ1 (pin 21 of MC9S12DP256) must be programmed as output and set for logic zero. If the board is used in expanded mode, the port B becomes the address/data bus, AD0-AD7, and the LEDs will add too much load on the bus. In order to make it work in expanded mode, J24A and J24B must be removed to disable the 7-segment LED display and the PB0-PB7 LEDs.

Port A is used as the 4X4 keypad interface in single chip mode, but in expanded mode, port A becomes the address/data bus AD8-AD15 and it cannot be connected with a keypad.

Port H is connected to an 8-position DIPswitch. The DIPswitch is connected to GND via the RN9 (eight 4.7K resistors), so it's not dead short to GND. When port H is programmed as an output port, the DIPswitch setting is ignored.

2.2 Hello Embedded World

Dragon12 Circuit Board Diagram



Dragon12_4.pdf

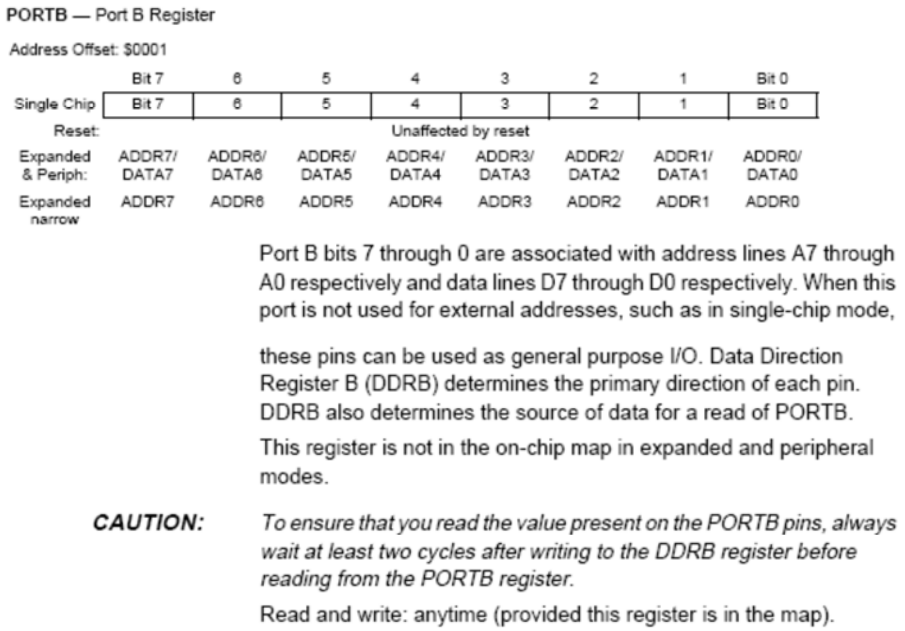
Dragon12_3.pdf

Dragon12_1.pdf

Step 2: Hardware setup of the microcontroller

Where are the I/O ports in the CPU’s memory address range and how to program them?

HCS12 docu 000-MC9S12DP256.pdf [3.0]: Register Map pg.66ff and pg.129ff Input/Output Registers



PORTB — Port B Register

Address Offset: $0001

	Bit 7	6	5	4	3	2	1	Bit 0
Single Chip	Bit 7	6	5	4	3	2	1	Bit 0
Reset:	Unaffected by reset							
Expanded & Periph:	ADDR7/ DATA7	ADDR6/ DATA6	ADDR5/ DATA5	ADDR4/ DATA4	ADDR3/ DATA3	ADDR2/ DATA2	ADDR1/ DATA1	ADDR0/ DATA0
Expanded narrow	ADDR7	ADDR6	ADDR5	ADDR4	ADDR3	ADDR2	ADDR1	ADDR0

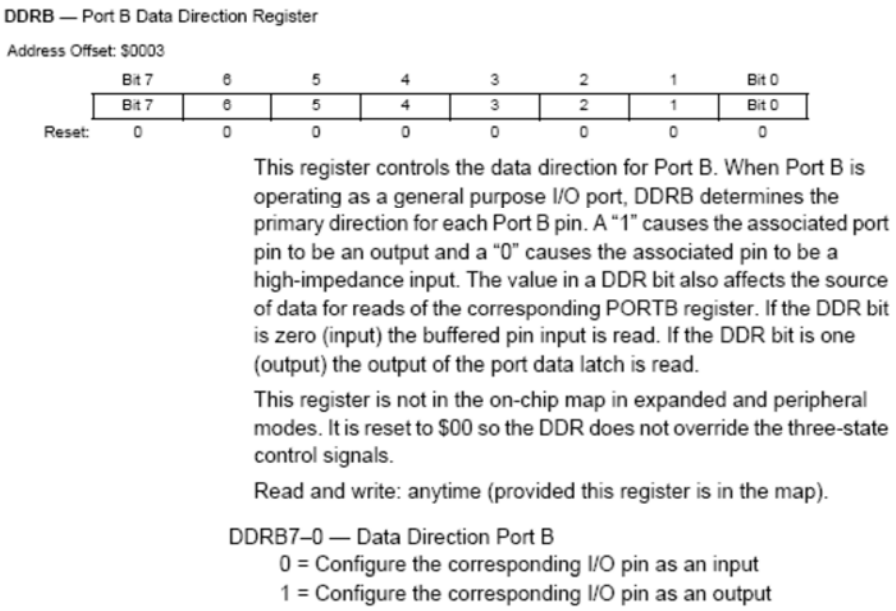
Port B bits 7 through 0 are associated with address lines A7 through A0 respectively and data lines D7 through D0 respectively. When this port is not used for external addresses, such as in single-chip mode,

these pins can be used as general purpose I/O. Data Direction Register B (DDRB) determines the primary direction of each pin. DDRB also determines the source of data for a read of PORTB.

This register is not in the on-chip map in expanded and peripheral modes.

CAUTION: *To ensure that you read the value present on the PORTB pins, always wait at least two cycles after writing to the DDRB register before reading from the PORTB register.*

Read and write: anytime (provided this register is in the map).



DDRB — Port B Data Direction Register

Address Offset: $0003

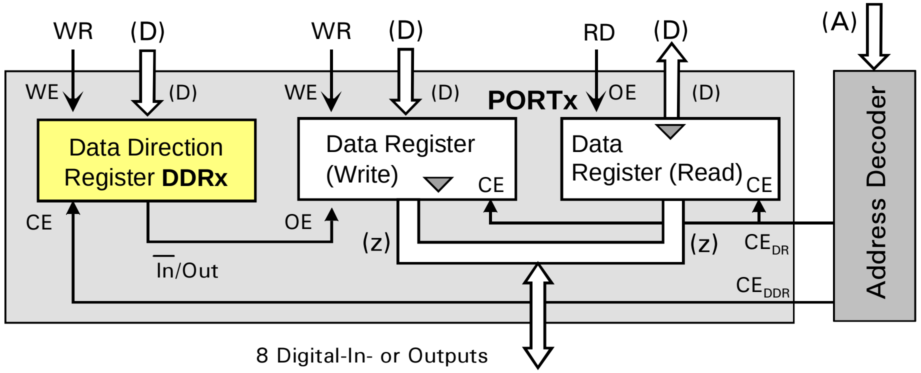
	Bit 7	6	5	4	3	2	1	Bit 0
	Bit 7	6	5	4	3	2	1	Bit 0
Reset:	0	0	0	0	0	0	0	0

This register controls the data direction for Port B. When Port B is operating as a general purpose I/O port, DDRB determines the primary direction for each Port B pin. A “1” causes the associated port pin to be an output and a “0” causes the associated pin to be a high-impedance input. The value in a DDR bit also affects the source of data for reads of the corresponding PORTB register. If the DDR bit is zero (input) the buffered pin input is read. If the DDR bit is one (output) the output of the port data latch is read.

This register is not in the on-chip map in expanded and peripheral modes. It is reset to $00 so the DDR does not override the three-state control signals.

Read and write: anytime (provided this register is in the map).

DDRB7–0 — Data Direction Port B
0 = Configure the corresponding I/O pin as an input
1 = Configure the corresponding I/O pin as an output



Data register PORTB at address $0001
Bit PB.x=1 → Port pin x = High

Data direction register DDRB at $0003
Bit DDRB.x=1 → Port pin x as output

Same concept for PORT J
Data register PTJ at address $0268
Data direction register DDRJ at address $026A

Step 3: Development environment, program design and coding

How do I write and compile a program?

- Installation and use of the IDE see **Appendix CodeWarrior**
- To simplify coding, instead of using hexadecimal addresses the IDE comes with include files defining symbols for registers and their respective bit positions, e.g.

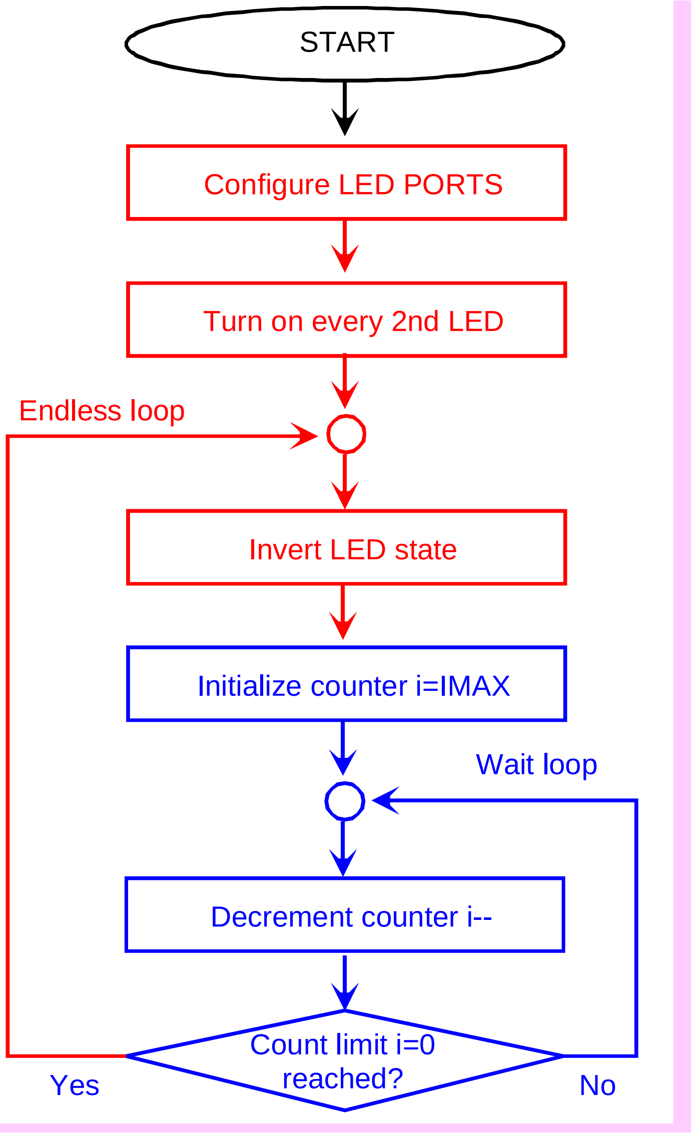
Predefined Symbols in Include-Files for	*for C-Programs* [*1]
	`mc9s12dp256.h`
		for Assembler-Programs
		`mc9s12dp256.inc`

Port B: Port	`#define` **`PORTB`** `(*(char*) 0x0001)`	**`PORTB`**`:    equ  $0001`
PORTB Bit 0	`#define` **`PORTB_BIT0`** `PORTB.Bits.BIT0`	
. . .	. . .	. . .
DDRB	`#define` **`DDRB`** `(*(char*) 0x0003)`	**`DDRB`**`:     equ  $0003`
DDRB Bit 0	`#define` **`DDRB_BIT0`** `DDRB.Bits.BIT0`	
Port J: Port	`#define` **`PTJ`** `(*(char*) 0x0268)`	**`PTJ`**`:      equ  $0268`
PTJ Bit 0	`#define` **`PTJ_PTJ0`** `PTJ.Bits.PTJ0`	
. . .	. . .	. . .
DDRJ	`#define` **`DDRJ`** `(*(char*) 0x026A)`	**`DDRJ`**`:     equ  $026A`
• DDRJ Bit 0	`#define` **`DDRJ_DDRJ0`** `DDRJ.Bits.DDRJ0`	

[*1] Simplified, actually ports are described via C-structures and unions of bit fields and byte or word-data types. Unfortunately, naming conventions for ports are not unified (**`PORTB`**, but **`PTJ`**)

2.2 Hello Embedded World

Design



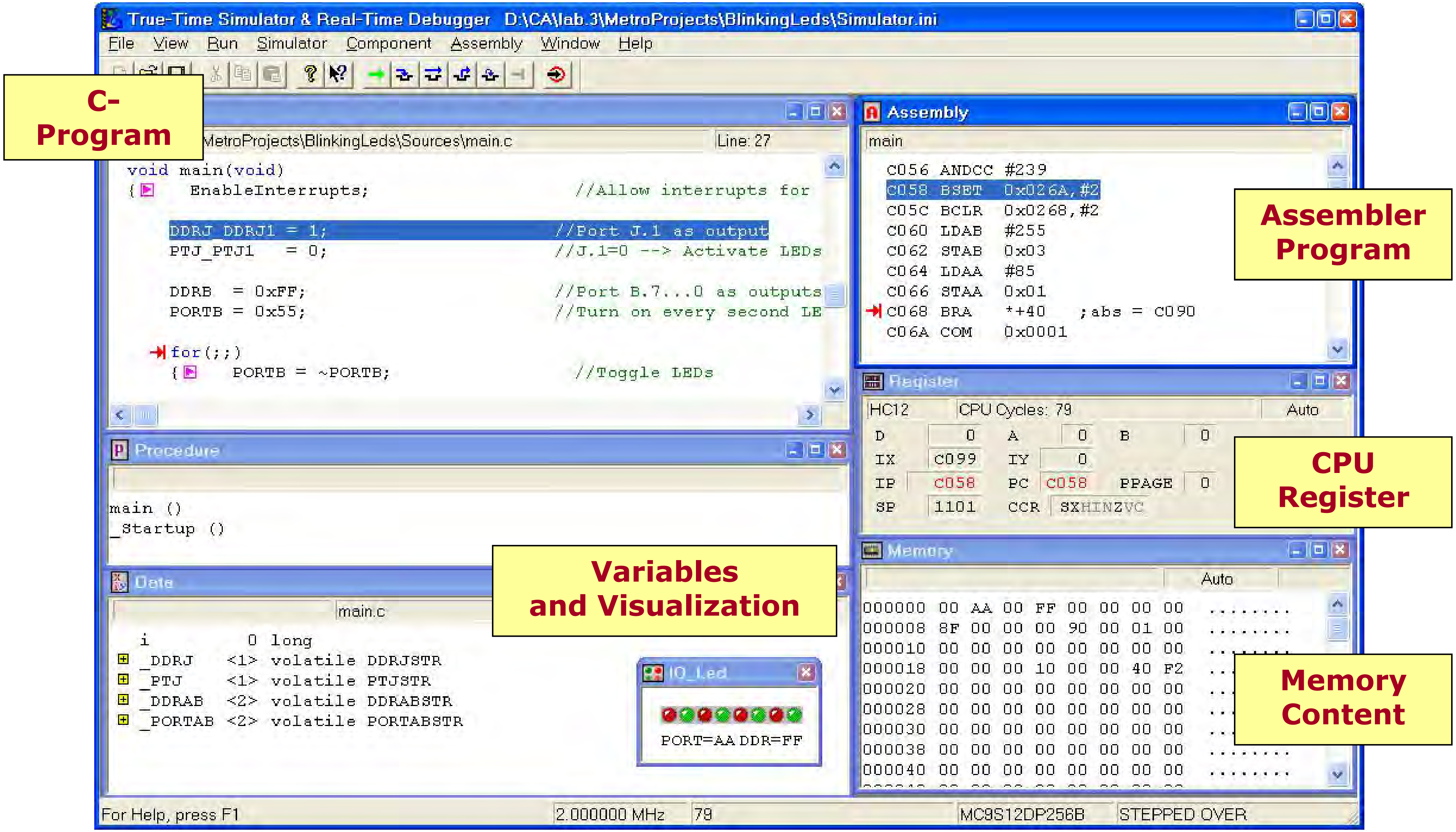
C-Code (CodeWarrior project **BlinkingLeds.mcp**)

```
#include <hidef.h>        //Common defines
#include <mc9s12dp256.h> //CPU specific defines

#pragma LINK_INFO DERIVATIVE "mc9s12dp256b"

#define IMAX  200000L    //Delay count

long i;                    //Counter variable

void main(void)
{   EnableInterrupts;      //Allow for debugger

    DDRJ_DDRJ1 = 1;       //Port J.1 as output
    PTJ_PTJ1   = 0;        //J.1=0 --> Activate LEDs

    DDRB  = 0xFF;          //Port B as outputs
    PORTB = 0x55;          //Turn on any other LED

    for(;;)
    {   PORTB = ~PORTB;    //Toggle LEDs (Bitwise Not)

        for (i=IMAX; i > 0; i--)
        {                 //Delay loop
        }
    }
}
```

PORT B on the Dragon12 board is connected to the LEDs and to the Seven-Segment-Display in parallel. If the Seven-Segment-Display shall not blink together with the LEDs, disable it by setting outputs Port P3….0= 1111_B

Step 4: Debug environment

How to download a program to the board and debug it? → see **Appendix CodeWarrior**


C-Program
Assembler Program
CPU Register
Variables and Visualization
Memory Content
True-Time Simulator & Real-Time Debugger D:\CA\lab.3\MetroProjects\BlinkingLeds\Simulator.ini
File View Run Simulator Component Assembly Window Help
Assembly
Register
Memory
Procedure
Data
IO_Led

MetroProjects\BlinkingLeds\Sources\main.c Line: 27

```
void main(void)
{   EnableInterrupts;                         //Allow interrupts for

    DDRJ_DDRJ1 = 1;                         //Port J.1 as output
    PTJ_PTJ1   = 0;                         //J.1=0 --> Activate LEDs

    DDRB   = 0xFF;                          //Port B.7...0 as outputs
    PORTB  = 0x55;                          //Turn on every second LE

    for(;;)
    {    PORTB  = ~PORTB;                     //Toggle LEDs
```

main

```
C056 ANDCC  #239
C058 BSET   0x026A,#2
C05C BCLR   0x0268,#2
C060 LDAB   #255
C062 STAB   0x03
C064 LDAA   #85
C066 STAA   0x01
C068 BRA    *+40      ;abs = C090
C06A COM    0x0001
```

```
HC12       CPU Cycles: 79                      Auto
D      0     A      0     B      0
IX  C099     IY     0
IP  C058     PC  C058     PPAGE  0
SP  1101     CCR SXHINZVC
```

```
main ()
_Startup ()
```

main.c

```
  i          0 long
⊞ _DDRJ    <1> volatile DDRJSTR
⊞ _PTJ     <1> volatile PTJSTR
⊞ _DDRAB   <2> volatile DDRABSTR
⊞ _PORTAB  <2> volatile PORTABSTR
```

```
PORT=AA DDR=FF
```

```
                                    Auto
000000  00 AA 00 FF 00 00 00 00   ........
000008  8F 00 00 00 90 00 01 00   ........
000010  00 00 00 00 00 00 00 00   ........
000018  00 00 00 10 00 00 40 F2   ...
000020  00 00 00 00 00 00 00 00   ...
000028  00 00 00 00 00 00 00 00   ...
000030  00 00 00 00 00 00 00 00   ...
000038  00 00 00 00 00 00 00 00   ........
000040  00 00 00 00 00 00 00 00   ........
```

For Help, press F1 | 2.000000 MHz | 79 | MC9S12DP256B | STEPPED OVER

2.2 Hello Embedded World

Memory Requirements of the C-Program (see file **Simulator.map**)

```
Summary of section sizes per section type:
READ_ONLY (R):          9B (dec: 155)      ← ROM: Program code + constant data
READ_WRITE (R/W):      104 (dec: 260)      ← RAM: Variable data 4 Byte (+ Stack 256 Byte *1)
NO_INIT (N/I):         23D (dec: 573)      ← Peripheral registers (fixed for all programs)
```

The C-compiler does not generate the most efficient code here. By manually programming in machine language (assembler code) rather than C, a faster and smaller program is possible (see next page)**:**

```
Summary of section sizes per section type:
READ_ONLY (R):          2A (dec:  42)      ← ROM: Program code + constant data
READ_WRITE (R/W):      100 (dec: 256)      ← RAM: Variable data 0 Byte (+ Stack 256 Byte *1)
                          . . .
```

Execution Speed (measured with HCS12 simulator)

Run time in CPU clock ticks	*C-Program*	*Assembler-Program*
CPU Reset to line Toggle LEDs	99 clocks	19 clocks
1 loop cycle Toggle LEDs to Toggle LEDs	47 clocks	17 clocks

Here the stack size could have been reduced even with the C-program. The assembler program could have been implemented completely without stack, if no debugging was required (the debug monitor uses the stack)

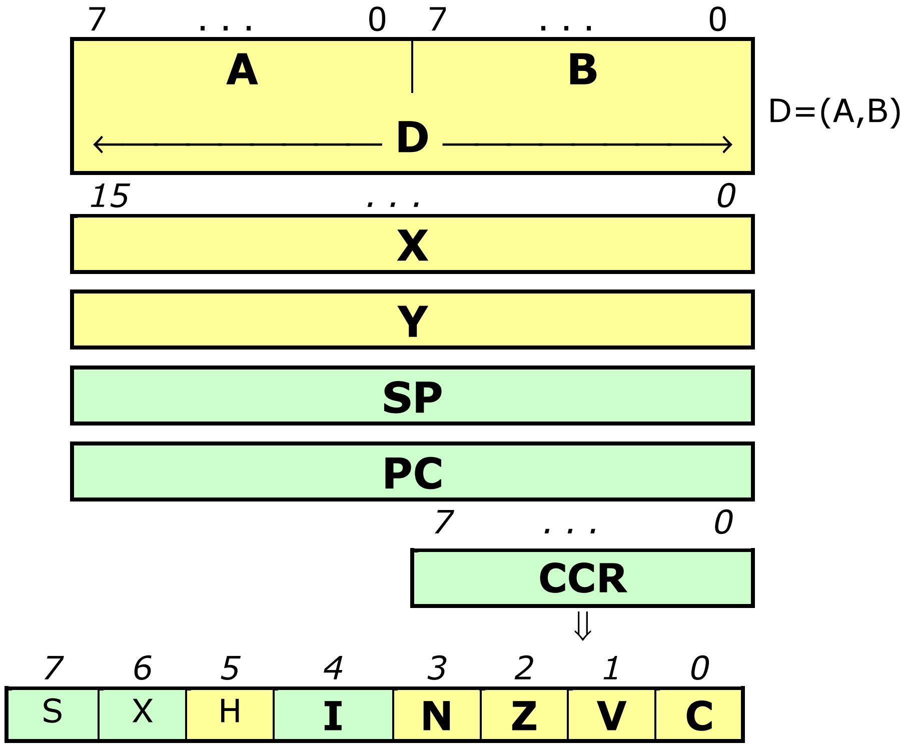
→ To understand, **how to optimize high level language programs** for embedded systems for size and speed (and to be able to find bugs in development tools like compilers and libraries) **knowledge of assembler language programming** is **required**.

2.2 Hello Embedded World

Blinking LEDs in optimized HCS12 Assembler (CodeWarrior project BlinkingLedsAsm.mcp)

```
     XDEF      Entry, main          ; Export symbols
     XREF      __SEG_END_SSTACK     ; Import symbols: End of stack
     INCLUDE   'mc9s12dp256.inc'    ; include derivative specific macros

IMAX: EQU 2048                      ; Symbolic constant: Delay count

.data:    SECTION                   ; RAM: Variable data section (not used in this program)

.const:   SECTION                   ; ROM: Constant data (not used in this program)

.init:    SECTION                   ; ROM: Code section
main:                               ; Begin of the program
Entry:    LDS  #__SEG_END_SSTACK    ; Initialize stack pointer
          CLI                       ; Enable interrupts, needed for debugger

          BSET DDRJ, #2             ; Bit Set:          Port J.1 as output
          BCLR PTJ,  #2             ; Bit Clear:        J.1=0 --> Activate LEDs

          MOVB #$FF, DDRB           ; $FF -> DDRB:      Port B.7...0 as outputs (LEDs)
          MOVB #$55, PORTB          ; $55 -> PORTB:     Turn on every other LED
loop:     COM  PORTB                ; Complement Port B:Toggle LEDs

          LDX  #IMAX                ; Delay loop to control toggle Frequency
wait0:    LDY  #IMAX                ; (Uses two nested counter loops with registers X and Y)
waitI:    DBNE Y, waitI             ; --- Decrement Y and branch to waitI if not equal to 0
          DBNE X, wait0             ; --- Decrement X and branch to wait0 if not equal to 0

          BRA loop                  ; Branch to loop
```

2.3 Register Model (registers accessible for assembler programs, see [3.1 Chapter 2])



Accumulator

16 bit register D, can be used in two halves as two 8 bit registers A and B for arithmetic-logic operations

Index Register X data and/or pointers

Index Register Y data and/or pointers

Stack Pointer SP pointer to stack

Program Counter PC address of next instruction (Instruction Pointer)

Condition Code Register (status register)

Status bits for arithmetic operations and control bits

CCR-Register Details

Carry =1	Overflow for operations with unsigned operands
Overflow =1	Overflow for signed operands (two’s complement)
Zero =1	Operation result is null
Negative =1	Operation result is negative
Interrupt mask =1	Disable on-chip interrupt signals (Reset: I=1)
Half carry	Overflow for operations with BCD-numbers
e**X**ternal interrupt mask	Disable external interrupt signals (Reset: X=1)
Stop disable	Ignore stop command (State after CPU reset: S=1)

2.3 Register Model, Data Types, Operand Addressing

Data Types

		HCS12		*80x86*
		Assembler [*1]	*HCS12 C* [*1]	*Visual C++*
• Natural numbers (unsigned) [*2] • Whole numbers (signed, 2s-complement)				

8bit	–128 … +127
	0 … 255
		`DC.B, DS.B` [*3]
		(byte)
			`char`
			`unsigned char`	

16bit	–32768 … +32767
	0 … 65535
		`DC.W, DS.W` [*3]
		(word)
			`short, int`
			`unsigned short`
				`short`
				`unsigned short`

32bit	-2147483648 …+2147483647
	0 … 4294967295
		`DC.L, DS.L` [*3]
		(long, dword)
			`long`
			`unsigned long`
				`int, long`
				`unsigned int, long`

• Floating-point numbers				
	IEEE 32bit	-	**`float, double`** [*1]	**`float`**
	IEEE 64bit	-	**`(double)`** [*1]	**`double`**
• Addresses/Pointers (for all data types)		16bit (near pointer)	16bit (near pointer)	32bit or 64bit
Bit field		1bit	8, 16 or 32bit	32bit
Enumeration		-	16bit	32bit
Array		[*3]	**`datatype name[count]`**	
Structure, union		-	**`struct, union`**	

Coding of Numbers and String Constants

		HCS12 Assembler	*C*
Decimal	(Base 10)	**`-34, 127`**	
Hexadecimal	(Base 16)	**`$3F8A , -$3F`**	**`0x3F8A`**
Octal	(Base 8)	**`@7345`**	
Dual	(Base 2)	**`%10101001`**	**`0b10101001`**
Floating-point		**`-`**	**`3.14159 , 1.6e-19`**
ASCII character		**`'Z'`**	**`'Z'`**
ASCIIZ string	*4	**`"This is a string", 0`**	**`"This is a String"`**

*1 The bit size of most data types can be configured via HCS12 C-compiler options.

*2 Assembler does not make a difference between **`signed`** and **`unsigned`** data.

*3 Variable in RAM-memory: **`name: DS.B count`**
defines 8bit variables in RAM-memory, which can be used via their **`name`**. The variables will **not** be initialized. Use **`count`** > 1 to define an array. Use **`DS.W`** and **`DS.L`** to define 16bit or 32bit variables and arrays.

Constant in ROM-memory: **`name: DC.B value`**
defines 8bit constants in ROM-memory, which can be used via their **`name`**. The constant will be initialized to **`value`** Use **`DC.W`** and **`DC.L`** to define 16bit and 32bit constants.

Use **`name: DCB.B count, value`**
to define a block of constants with **`count`** bytes and initialize each byte to **`value`**. Same with **`DCB.W`** and. **`DCB.L`**.

*4 In C the end of a string will automatically be marked with a 0-byte (ASCII Zero String). In Assembler the 0-byte must be specified explicitly.

Operand Address Modes (see [3.1 Chapter 3], [1.4 Chapter 4], [2.1 Chapter 2.7])

HCS12 is a **Two-Address-CPU**, i.e. a CPU instruction can have up to two operands. One of the operands (destination operand) will be overwritten by the instruction’s result:

Register operands in C/C++ in HCS12-ASM

(Explicit) **Register Address**	**`INST reg`** Registers are explicitly specified as operands. Rarely used, HCS12 prefers implicit register addresses.	
Example: X=D (D→X)	TFR D, X	Copy value of register D to register X

Implicit **(Register) Address** Freescale term: Inherent INH	**`INST`** The operand (one of the registers A, B, D, X, Y, SP) is contained implicitly in the instruction mnemonic.	
Example: X++	INX	Increment the value of register X

Memory variable operand

Direct Address DIR 8bit, EXT 16bit Address	**`INST address`** The operand’s memory address is part of the instruction. Programmers typically use variable names rather than addresses The address will be assigned by the compiler.	
Example: D=*0x2000 D = var1	LDD $2000 (no Hash) LDD var1	Load D with the value at memory address 2000_h Load D with the value of variable var1

2.3 Register Model, Data Types, Operand Addressing

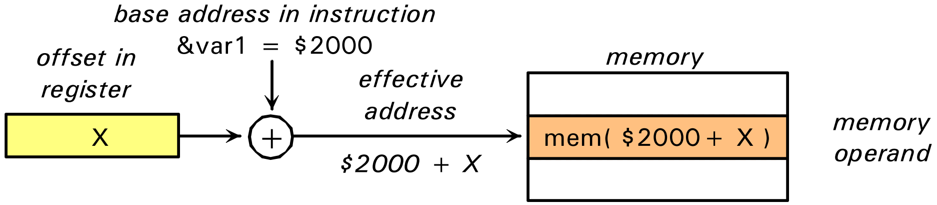
Constant operands Warning: Without #… **`const`** would be an address rather than a constant!

Immediate Operand Immediate IMM	**`INST #const`** The operand is part of the instruction. Constants must be marked by **#…**, e.g. #20, #–20, #$0A, #%01101011	
Example: D=0xB010	LDD #$B010 (with Hash!)	Load constant $B010_h$ into register D immediate for source operand, dest. operand implicit
D= &var1	LDD #var1	Load D with the address of variable var1

Indirect Address in various variants (Motorola/Freescale term: Indexed)		
Register-indirect … Indexed IDX	**`INST 0, reg`**$_{X,Y,SP}$ Memory address in register X, Y, SP, i.e. register used as pointer	
Example: D = *X	LDD 0, X	Load register D with the value at memory address in X (**indirect address**)
… with Pre- or Post-Increment or Decrement Auto Increment IDX	**`INST const`**$_{1,\dots,+8}$**`, {+|-}reg`**$_{X,Y,SP}$ **`INST const`**$_{1,\dots,+8}$**`, reg`**$_{X,Y,SP}$**`{+|-}`** The pointer in register X, Y, SP will be incremented or decremented by some constant 1, … , 8 before (pre) or after (post) using the pointer to address the operand.	
X = X – 2; D =*X	LDD 2, –X	Load memory value, to which X points, into register D, decrement X by 2 **before**
D = *X; X = X + 4	LDD 4, X+	. . ., increment X by 4 **afterwards**

2.3 Register Model, Data Types, Operand Addressing

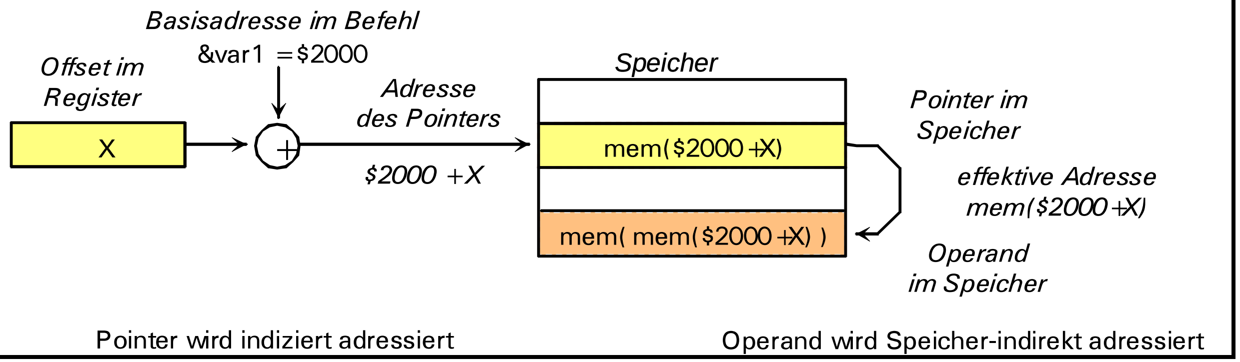
… with index/offset Indexed IDX 5bit constant IDX1 9bit constant IDX2 16bit constant	**INST const, reg$_{X,Y,SP,PC}$** address = const +reg$_{X,Y,SP,PC}$ **INST reg$_{A,B,D}$, reg$_{X,Y,SP,PC}$** address = reg$_{A,B,D}$+reg$_{X,Y,SP,PC}$ The operand's address is the sum of a constant plus register X, Y or SP or the sum of two registers A, B or D plus X, Y or SP.	

Example:
Let char var1[…]
with &var1=0x2000
Y = var1[X]
var1[X] = *(&var1+X)
Y = *(D+X)
	← array of char
	LDY var1, X
	(instead of var1 any 16bit constant possible)
	LDY D, X
		Load Y with var1[X], i.e. the value at memory address var1 + X (indexing array var1 by index X)
		
		Load Y with contents of memory address D + X

Memory-indirect **… with index** Indexed-Indirect [IDX2]	**INST [const, reg$_{X,Y,SP,PC}$]** **INST [D, reg$_{X,Y,SP,PC}$]** The operand's memory address is in a pointer in memory. This memory address will be addressed via another pointer, which is calculated as the value of register X, Y, SP or PC plus a constant or register D (Note: A, B not allowed here!).	
Example: Y = *(*(D+X))	LDY [D, X]	Load Y with the value of the memory cell, to which the memory pointer points, to which D+X point

2.3 Register Model, Data Types, Operand Addressing

Let char *var1[…] With &var1[…]=0x2000 Y = *var1[X] *var1[X] = *(*(&var1+X))	← Array of pointers to char LDY [var1, X]	Load Y with the memory cell to which a pointer in var1[X] points. (access via an array of pointer)



Branch instructions use so called **relative addressing** (Motorola/ Freescale-name REL). Relative addresses use the current value of the instruction pointer and add a constant offset, which is included in the instruction. The programmer need not care about details, but simply uses a label as the target of the branch:

```
start:  . . .
        BRA start
```

Unfortunately, normal branches limit the offset to 8bit. However, there is a Long Branch version **LBRA** using a 16bit offset (see chapter 2.6).

2.4 Instruction Set 1: Data Transport (see [2.1 Chapter 2.9], [3.1 Chapter 5, 6])

- Data transport instructions (including stack)
- Arithmetic and logic instructions
- Compare and branch instructions (including software interrupts)
- Miscellaneous instructions

Abbreviations

reg$_{\mathbf{A,B,D}}$**. . .**	One of the registers **A**, **B**, **D**, **. . .**
mem	Memory operand with arbitrary memory addressing (direct, indexed, indirect-indexed)
imm	Immediate operand
mem_i	**mem** or **imm**
adr	Code address relative to **PC**
LD{AA|AB|…|S}	Abbreviation for **LDAA**, **LDAB** or **LDS**
8bit or **16bit**	Used as index: Size of an operand

If not stated otherwise, all instructions do modify CCR status bits N, Z, V, C depending on the instruction’s result such that conditional branch instructions can be used directly without a preceding compare instruction.

2.4 Data Transport Instructions

Transport Instructions

(Status bits N, Z, V, C are modified by LD… and ST… instructions only)

LD{AA|AB|D|X|Y|S} mem_i *2	**mem_i → reg$_{A,B,D,X,Y,SP}$**	LoaD register from memory A, B are loaded with an 8bit, D, X, Y, SP are loaded with an 16bit value
ST{AA|AB|D|X|Y|S} mem *2	**reg$_{A,B,D,X,Y,SP}$ → mem**	STore register to memory
TFR reg$_{A,B,D,X,Y,SP,CCR}$ **,** **reg**$_{A,B,D,X,Y,SP,CCR}$ *1	**reg → reg**	TransFeR register to register If the source register is 8bit and the destination register is 16bit, the MS-Byte will be loaded with the sign of the 8bit value (Sign Extension). Vice versa only the LSByte will be copied.
EXG reg$_{A,B,D,X,Y,SP,CCR}$ **,** *1 **reg**$_{A,B,D,X,Y,SP,CCR}$	**reg ↔ reg**	EXchanGe register Swap register contents
TAB, TBA **TSX, TSY, TXS, TYS** **TAP, TPA** **XGDX, XGDY**	**A → B** bzw. **B → A** **SP→X, SP→Y, X→SP, Y→SP** **A → CCR, CCR → A** **D ↔ X, X ↔ D**	Variants of TFR and EXG (shorter opcodes)
MOVB mem_i, mem **MOVW mem_i, mem** *1	**mem_i → mem** 8bit **mem_i → mem** 16bit Adressing [IDX], IDX1, IDX2 not possible, for IDX only -16 … +15 is allowed	MOVe Byte MOVe Word Direct memory to memory copy
SEX reg$_{A,B,CCR}$**, reg**$_{D,X,Y,SP}$ *1	**reg$_{A,B,C}$ → reg$_{D,X,Y,SP}$**	Sign EXtension Copy from 8bit to 16bit for 2s-complement-numbers (same as TFR)

[*1] These instruction do not modify status bits N, Z, V, C.

[*2] These instructions do modify status bits N, Z, V, but not C !

2.4 Data Transport Instructions

Calculate a pointer (indexed or indirect address = effective address)

LEA{X|Y|S} mem *1	Address of **mem** → **reg**$_{\mathbf{X,Y,SP}}$	Load Effective memory Address into register Note: Calculation is done during run-time, not compile time

Stack (see chapter 2.6)

PSH[A|B|C] *1 **PSH{D|X|Y}** *1	**SP-1→SP, reg**$_{\mathbf{A,B,CCR}}$**→Stack** **SP-2→SP, reg**$_{\mathbf{D,X,Y}}$**→Stack**	PuSH register to stack Copy register on stack
PUL[A|B|C] *1 **PUL{D|X|Y}** *1	**Stack→reg**$_{\mathbf{A,B,CCR}}$**, SP+1→SP** **Stack→reg**$_{\mathbf{D,X,Y}}$**, SP+2→SP**	Pull register from stack Copy from stack to register

*1 These instructions (except **PULC**) do not modify status bits N, Z, V, C.

Note:
PSH… und **PUL…** can be substituted by **ST…** and **LD…**:

E.g.:

```
STAA 1, -SP     =   PSHA
STD  2, -SP     =   PSHD

LDAA 1, SP+     =   PULA
LDD  2, SP+     =   PULD
```

Modification of the stack pointer **SP** without actually copying data (required in chapter 4):

LEAS n, -SP Allocate stack space for n byte (LEAS n, –SP = LEAS –n, SP)
LEAS n, +SP Free n Byte from stack (LEAS n, SP = LEAS n, +SP)

2.4 Data Transport Instructions

Example Program 1

(CodeWarrior project AsmIntro.mcp)

```
.data: SECTION              ; Global variables in RAM (uninitialized)
var1:    ds.w   1           ; short var1
var2:    ds.b   1           ; char var2
var3:    ds.b   2           ; char var3[2]

.const: SECTION             ; Global constants in ROM (initialized)
const1: dc.b   $00, $11, $22, $33    ; const char const1[4]= { 0x00, 0x11, ... };

.init:  SECTION             ; Program code      (Address mode of explicit operand)
main:    . . .
        LDD   #$1234        ; D = 0x1234                  (immediate)
        TFR   D, X          ; X = D = 0x1234              (register)
        STD   var1          ; var1 = D = 0x1234           (direct)
        STAA  var2          ; var2 = A = 0x12
        STD   var3          ; var3 = D  with  var3[0] = 0x12,  var3[1]= 0x34

        LDD   const1        ; D = const1 = 0x0011         (direct)
        LDD   #const1       ; D = &const1                 (immediate)
```

2.4 Data Transport Instructions

```
Continued              ; D=&const1 (from previous instruction)

       LDY  #$0001     ; Y = 0x0001                                   (immediate)
       LDX  D, Y       ; X = *(D+Y) = *(&const1+1) = 0x1122  (indexed)
       LDX  const1, Y  ; X = const1[Y] = 0x1122          (indexed)  const1[Y]=*(&const1+Y)
       LDY  #const1    ; Y = &const1
       LDAA 1, Y+      ; A = *Y = 0                      (indirect with post-increment)
                       ; Y = Y+1 = &const1[1]
       LDAA 2, +Y      ; Y = Y+2 = &const1[3]
                       ; A = *Y =  0x33                  (indirect with pre-increment)
       LDAA 1, -Y      ; Y = Y–1 = &const1[2]
                       ; A = *Y = 0x22                   (indirect with pre-decrement)
       LDAA 1, Y-      ; A = *Y = 0x22                   (indirect with post-decrement)
                       ; Y = Y–1 = &const1[1]
       LDD  #const1    ; D = &const1
       STD  var1       ; var1 = D = &const1
       LDX  #0000      ; X = 0
       LDD  var1, X    ; D = var1[X] = *(&var1+X) = &const1    (indexed)
       LDD  [var1, X]  ; D = *var1[X]= *(*(&var1+X))= 0x0011(indirect indexed)
```

2.4 Data Transport Instructions

Continued

```
LDD  #$AAAA      ; D = 0xAAAA
LDX  #$5555      ; X = 0x5555
LDAA #$7F        ; A = 0x7F
TFR  A, X        ; X = A = (sign extended) 7Fh  = 0x007F
LDAA #$80         ; A = 0x80
TFR  A, X        ; X = A = sign extended 80h = 0xFF80
TFR  X, B        ; B = X$_{LSB}$ = 0x80

MOVW #$5678,var1; var1 = 0x5678
MOVW var1, var2 ; var2 =0x56, but note: var3[0] = 0x78 will be overwritten
LDX  #var3       ; X = &var3
MOVB var1, 0,X  ; *X = var3[0] = MSB of var1 = 0x56
MOVB 0,X,  1,X  ; var3[1] = var3[0] = 0x5656

LDD  var1        ; D = var1 = 0x5678
LDD  var1+1      ; D = (LSByte of var1, MSByte of var2) = 0x7856
LDD  var1+3      ; D = (var3[0], var3[1]) = 0x5656
```

2.5 Instruction Set 2: Arithmetic and Logic Operations

Add, Subtract, Increment, Decrement, Invert Sign

AB{A|X|Y} **SBA**	**B+A → A, B+X → X, B+Y → Y** **A –B → A**	ADD/SuBtract (A, B are loaded with a 8bit, D, X, Y, are loaded with a 16bit value)
ADD{A|B|D} mem_i **SUB{A|B|D} mem_i**	**reg**$_{A,B,D}$ **+ mem_i → reg**$_{A,B,D}$ **reg**$_{A,B,D}$ **– mem_i → reg**$_{A,B,D}$	ADD 8bit ±8bit or 16bit ±16bit SUBtract
ADC{A|B} mem_i$_{8bit}$ **SBC{A|B} mem_i**$_{8bit}$ (**ADC**, **SBC** not with D)	**reg**$_{A,B}$ **+ mem + C → reg**$_{A,B}$ **reg**$_{A,B}$ **– mem – C → reg**$_{A,B}$	ADd with Carry 8bit SuBtract with Carry 8bit
INC mem$_{8bit}$ **IN{CA|CB|X|Y|S}** [*1] **DEC mem**$_{8bit}$ **DE{CA|CB|X|Y|S}** [*1] (**INC**, **DEC** not with D)	**mem+1 → mem** **reg**$_{A,B,X,Y,S}$**+1→ reg**$_{A,B,X,Y,S}$ **mem-1 → mem** **reg**$_{A,B,X,Y,S}$**-1 → reg**$_{A,B,X,Y,S}$	INCrement memory 8bit INcrement register DECrement memory 8bit DEcrement register
CLR mem$_{8bit}$ **CLR{A|B}** (**CLR** not with D)	**0 → mem** **0 → reg**$_{A,B}$	CLeaR byte (Load with 0)
NEG mem$_{8bit}$ **NEG{A|B}** (**NEG** not with D)	**–mem → mem** **–reg**$_{A,B}$ **→ reg**$_{A,B}$	NEGate byte Multiply by –1 (invert sign), sets C=1 if A≠0 or B≠0, sets V, if A=\$80 or B=\$80 !

[*1] INS und DES do not modify status bits N, Z, V, C.

2.5 Arithmetic and Logic Instructions

Bitwise logical Operations

COM mem$_{8bit}$	**/mem → mem**	COMplement 1’s-complement-
COM{A|B}	**/reg$_{A,B}$ → reg$_{A,B}$**	(Bitwise NOT)
AND{A|B} mem_i$_{8bit}$	**reg$_{A,B}$ AND mem _i → reg$_{A,B}$**	Bitwise AND
ANDCC imm $_{8bit}$	**CCR AND imm → CCR**	
ORA{A|B} mem_i$_{8bit}$	**reg$_{A,B}$ OR mem_i → reg$_{A,B}$**	Bitwise OR
ORCC imm$_{8bit}$	**CCR OR imm → CCR**	
EOR{A|B} mem_i$_{8bit}$	**reg$_{A,B}$ XOR mem_i → reg$_{A,B}$**	Bitwise Exclusive OR

Bit Operations

CLC, SEC	**0 → C, 1 → C**	CLear/SEt Carry bit in CCR
CLV, SEV	**0 → V, 1 → V**	CLear/SEt oVerflow bit in CCR
BCLR mem$_{8bit}$, imm	**mem AND /imm → mem**	Bit ClearR 8bit
BSET mem$_{8bit}$, imm	**mem OR imm → mem**	Bit SET 8bit

Multiply, Divide

MUL	**A x B → D** unsigned	MULtiply 8bit x 8bit → 16bit
EMUL, **EMULS**	**D x Y → (Y, D)** unsigned/signed	16bit x 16bit → 32bit
IDIV, IDIVS	**D / X → X, Remainder in D**	DIVide 16bit / 16bit → 16bit
EDIV, EDIVS	**(Y, D)/ X → Y, Rem. in D** unsigned/signed	32bit / 16bit → 16bit
FDIV	**D*2^{16} / X → X, Rem. in D**	“Pseudo 32bit” / 16bit →16bit

2.5 Arithmetic and Logic Instructions

Shift and Rotate

LSL mem$_{8bit}$
LSL{A|B|D}
ASL mem$_{8bit}$
ASL{A|B|D}
	mem << 1 → mem 8bit
	reg$_{A,B,D}$ << 1 → reg$_{A,B,D}$
		Logical Shift Left
		Arithmetic Shift Left
		Shift left by 1bit for signed and unsigned values
		MSB shifted to CCR Carry bit
		LSB cleared to 0

LSR mem$_{8bit}$ LSR{A|B|D}	mem >> 1 → mem 8bit reg$_{A,B}$ >> 1 → reg$_{A,B,D}$	Logical Shift Right Shift right by 1bit for unsigned values. LSB shifted to CCR Carry Bit MSB cleared to 0
ASR mem$_{8bit}$ ASR{A|B} (ASR not with D)	mem >> 1 → mem 8bit reg$_{A,B}$ >> 1 → reg$_{A,B,D}$ (MSB=Vorzeichen bleibt unverändert)	Arithmetic Shift Right Shift right by 1bit for signed values. LSB shifted to CCR Carry Bit. MSB (=sign of value) not changed.
ROL mem$_{8bit}$ ROL{A|B} (ROL not with D)	mem << 1 → mem + C 8bit reg$_{A,B}$ << 1 → reg$_{A,B}$ + C	ROtate Left Rotate left by 1bit. Carry Bit shifted to LSB, MSB shifted to Carry Bit.
ROR mem$_{8bit}$ ROR{A|B} (ROR not with D)	mem >> 1 → mem + C * 8 reg$_{A,B}$ >> 1 → reg$_{A,B}$ + C * 8	ROtate Right Rotate right by 1bit. Carry Bit shifted to MSB, LSB shifted to Carry Bit.

2.5 Arithmetic and Logic Instructions

Example Program 2 (CodeWarrior project AsmIntro2.mcp)

C-Program	*Equivalent Assembler-Program*

`char a08 = 1, c08 = 3;`
`int  a16 = 1, b16 = 2, c16 = 3;`
`long a32 = 1, b32 = 2, c32 = 3;`
`unsigned char cu08= 3;`
`unsigned int  cu16= 3;`	

`void main(void)`
`{   c16 = a16 + b16;` //Addition 16bit
	`LDD   a16`
	`ADDD  b16`
	`STD   c16`

`c32 = a32 + b32;` //Addition 32 bit	`LDD   a32+2     ;Add LSW`
	`ADDD  b32+2`
	`STD   c32+2`
	`LDD   a32       ;Add MSW`
	`ADCB  b32+1     ;     with Carry!`
	`ADCA  b32       ;(HCS12 has no 16bit ADC)`
	`STD   c32`

`c08 = (char) c16;` //signed 16 → 8bit	`LDAB  c16+1     ;LSByte of c16`
	`STAB  c08       ;MSbyte not used`
	`(alternative: MOVB c16+1, c08)`

`cu08 = (unsigned char) cu16;`
//unsigned 16 → 8bit
	`LDAB  cu16+1    ;same as signed`
	`STAB  cu08`

2.5 Arithmetic and Logic Instructions

C-Program	*Equivalent Assembler-Program*
c16 = c08; //signed 8 → 16 bit	LDAB c08 SEX B,X ;Sign extension STX c16
cu16 = cu08; //unsigned 8 → 16 bit	MOVB cu08, cu16+1 ; LSB MOVB #0, cu16 ;Set MSB=0
cu16= cu16 >> 2; //Shift right unsigned	LDD cu16 LSRD ;2x shift logic LSRD STD cu16
c16= c16 >> 2; //Shift right signed	LDD c16 ASRA ;Shift MSByte arith. ; with LSB → CY RORB ;Shift LSByte ;fill LSB from MSByte ASRA ;same again ... RORB STD c16
c08 = c08 | 0x81; //Set bits 7 and 0 to ‘1’	BSET c08, #$81
a08 = a08 & ~0x81;//Set bits 7 and 0 to ‘0’	BCLR a08, #$81

2.5 Arithmetic and Logic Instructions

C-Program	*Equivalent Assembler-Program*

`c16 = a16 ^  b16;` //Bitwise Exclusive OR	`LDD   a16`
	`EORB  b16+1 ;(HCS12 has no 16bit EOR)`
	`EORA  b16`
	`STD   c16`

`c16 = a16 &  b16;` //Bitwise AND	`LDD   a16   ;as above, but AND`
	`ANDB  b16+1 ; instead of EOR`
	`ANDA  b16`
	`STD   c16`

`c16 = a16 && b16;` //Logical AND	`    LDD   a16   ;(AND etc. work bitwise!)`
	`    CPD   #0    ;a16==FALSE (0) ?`
	`    BEQ   L1`
	`    LDD   b16`
	`    CPD   #0    ;b16=FALSE (0) ?`
	`    BNE   L2`
	`L1: LDY   #0    ;Result FALSE (0)`
	`    BRA   L3`
	`L2: LDY   #1    ;Result TRUE  (1)`
	`L3: STY   c16`

2.6 Instruction Set 3: Compare and Branch

Compare and Test

CBA **CMP{A|B} mem_i**$_{\mathbf{8bit}}$ **CP{D|X|Y|S} mem_i**$_{\mathbf{16bit}}$	Compute **A – B** Compute **reg**$_{\mathbf{A,B}}$ **- mem_i** Compute **reg**$_{\mathbf{D,X,Y,SP}}$**- mem_i**	Compare Compare register with register, variables or constant, set bits in CCR
TST mem$_{\mathbf{8bit}}$ **TST{A|B}**	Compute **mem – 0** Compute **reg**$_{\mathbf{A,B}}$ **– 0**	Test if operand is 0 or negative, set bits in CCR
BIT{A|B} mem_i$_{\mathbf{8bit}}$	Compute **reg**$_{\mathbf{A,B}}$ **AND mem_i**	BIt Test Like AND, but sets CCR bits only

Unconditional and Conditional Branches (branches check, but do not change status bits N, Z, V, C)

JMP mem	**mem** → **PC**		JuMP like {L}BRA, but can use indirect/indexed addressing
{L}BRA adr	**adr** → **PC**		BRanch Always
{L}BRN adr	No Operation, same as NOP		BRanch Never
{L}BCC adr	**adr** → **PC**,	if C=0	Branch if Carry Clear
{L}BCS adr	. . .	if C=1	Branch if Carry Set
{L}BNE adr	. . .	if Z=0	Branch if Not Equal
{L}BEQ adr	. . .	if Z=1	Branch if EQual
{L}BPL adr	. . .	if N=0	Branch if Plus (positive)
{L}BMI adr	. . .	if N=1	Branch if Minus (negative)
{L}BVC adr	. . .	if V=0	Branch if Overflow Clear
{L}BVS adr	. . .	if V=1	Branch if Overflow Set

2.6 Compare and Branch Instructions

{L}BGT adr	**adr** → **PC** if . . . >	Branch if GreaTer
{L}BGE adr	>=	Branch if Greater or Equal
{L}BEQ adr	==	Branch if Equal
{L}BLE adr	<=	Branch if Less or Equal
{L}BLT adr	<	Branch if Less Use after a compare or arithmetic operation with **signed** values
{L}BHI adr	**adr** → **PC** if . . . >	Branch if Higher
{L}BHS adr	>=	Branch if High or Same
{L}BEQ adr	==	Branch if Equal
{L}BLS adr	<=	Branch if Lower or Same
{L}BLO adr	<	Branch if Lower Use after a compare or arithmetic operation with **unsigned** values
BRCLR mem$_{8bit}$**, imm, adr**	**adr**→**PC** if **mem & imm=0**	BRanch if bits are CLeaRed
BRSET mem$_{8bit}$**, imm, adr**	**adr**→**PC** if **/mem & imm=0**	BRanch if bits are SET

All conditional branches check the status bits in the CCR register, which have been set by a previous operation, typically a compare.

adr is a code memory address, which the programmer did specify via a label. The instruction uses relative addressing (see chap. 2.3). Normal branches with 8bit offsets can only jump +/ 128 bytes from the current instruction pointer location. If you need to jump over a longer distance, use the long branch instructions **{L},** which use 16bit offsets and thus can reach any HCS12 address.

2.6 Compare and Branch Instructions

Loop Instructions (These instructions do not modify status bits N, Z, V, C)

IBEQ reg$_{A,B,D,X,Y,SP}$**, adr** **DBEQ reg**$_{A,B,D,X,Y,SP}$**, adr** **IBNE reg**$_{A,B,D,X,Y,SP}$**, adr** **DBNE reg**$_{A,B,D,X,Y,SP}$**, adr**	**reg**$_{A,B,D,X,Y,SP}$ ± 1 → **reg**$_{...}$ **adr** → **PC** if **reg**$_{...}$ **= 0** **adr** → **PC** if **reg**$_{...}$**!= 0**	Increment/Decrement register and . . . . . . Branch if Equal to 0 . . . Branch if Not Equal to 0
TBEQ reg$_{A,B,D,X,Y,SP}$**, adr** **TBNE reg**$_{A,B,D,X,Y,SP}$**, adr**	**adr** → **PC** if **reg**$_{...}$ **= 0** if **reg**$_{...}$**!= 0**	Test register and Branch if …

Example Program 3 (CodeWarrior project AsmIntro2.mcp)

C-Program	*Equivalent Assembler-Program*
`if (c16 <= 32)` //if – else	`LDD c16`
	`CPD #32 ;Compare`
	`BGT L1 ;Assume signed numbers`
`{ a08 = 4;`	`MOVB #4, a08 ;or LDAB #4; STAB a08`
`. . .`	`. . .`
	`BRA L2`
`} else`	
`{ a08 = 8;`	`L1: MOVB #8, a08 ;or LDAB #4; STAB a08`
`. . .`	`. . .`
`}`	
`. . .`	`L2: . . .`
`if (cu16 <= 32)` //if - else	`LDD cu16`
	`CPD #32 ;Compare`
	`BHI L3 ;Assume unsigned numbers`
`{ . . .`	`. . .`
`}`	
`. . .`	`L3: . . .`
`for (;;);` //endless loop	`BRA *+0 ;* =current location counter`

2.6 Compare and Branch Instructions

C-Program	*Equivalent Assembler-Program*

for (c08=0; c08 < 3; c08++) //for
{ c16 = c16 + a16;
}
	CLR c08 ;Initialize loop counter
	BRA L4 ;Jump to test condition
	L0: LDD c16 ;Loop body {…}
	ADDD a16
	STD c16
	L1: INC c08 ;Increment loop counter
	L4: LDAB c08 ;Test loop condition
	CMPB #3
	BLT L0 ;Go to next step

while (c08 <= 32) //while - do
{ a16++;
}
	BRA L3 ;Branch to test cond.
	L2: LDX a16 ;Loop Body {…}
	INX
	STX a16
	L3: LDAB c08 ;Test loop condition
	CMPB #32
	BLE L2 ;Go to next step

do { ... } while (c08 <= 32) //do - while	Same as while, but without **BRA L3**

2.6 Compare and Branch Instructions

C-Program	*Equivalent Assembler-Program*
	NONE: EQU 0 ; Values of the ONE: EQU 1 ; enumeration TWO: EQU 2
enum { NONE, ONE, TWO } eVal;	eVal: DS.W 1 ; Enumeration
. . .	. . .
switch (eVal) //switch-case	switch: LDD eVal ; Compute LSLD ; index into TFR D, X ; branch table JMP [swK, X]
	swK: DC.W caseNONE ; Branch DC.W caseONE ; table DC.W caseTWO
{ case NONE: . . .	caseNONE: . . .
break;	BRA endCase
case ONE: . . .	caseONE: . . .
break;	BRA endCase
case TWO: . . .	caseTWO: . . .
break;	BRA endCase
}	endCase:

2.7 Instruction Set 4: Subroutine Calls and State Management

Subroutine Calls (These instructions do not modify status bits N, Z, V, C)

JSR mem	**mem → PC** Saves return address on stack	Jump to SubRoutine Like BSR, but can use indirect/indexed destination address
{L}BSR adr	**adr → PC** Saves return address on stack	Branch to SubRoutine Like JSR, but only relative addresses (shorter opcode than JSR)
RTS	Restores the return address from stack	ReTurn from SubRoutine
CALL, RTC	Subroutine call and return for memory sizes > 64KB	

Interrupts (see chapter 3) (These instructions do not modify status bits N, Z, V, C)

Interrupt = subroutine, which will be called by a hardware event. An interrupt will store registers on the stack.

RTI	Restores registers from stack. Do not use RTS for interrupts!	ReTurn from Interrupt
CLI	0 → I (Note: Debugger shows this as ANDCC #$EF)	CLear Interrupt mask Global interrupt enable.
SEI	1 → I (Note: Debugger shows this as ORCC #$10)	SeT Interrupt mask Global interrupt disable.
SWI	Store return address and register set X, Y, D, CCR on stack, not maskable, does disable interrupts I=1	SoftWare Interrupt Call the SWI Interrupt Service Routine (used by debug monitor)

2.7 Subroutine Calls and CPU State Management

TRAP	Like SWI	TRAP for unimplemented op-codes Call the TRAP Interrupt Service Routine

Miscellaneous Operations

NOP	-	No Operation
WAI, STOP	WAIt and STOP Energy Saving mode: Turn CPU off without/with all on-chip peripherals. Operation will be resumed via an interrupt. Should not be used when testing a program with the HCS12 debugger.	
MEM, REV, EMIN…, EMAX…, MIN…, MAX…, ETBL, TBL, …	Instructions to implement Fuzzy Logic minimum and maximum operations and data table access see [3.1].	

2.7 Subroutine Calls and CPU State Management

Example Program 4 (CodeWarrior project AsmIntro2.mcp)

C-Program	*Equivalent Assembler-Program*
`int betrag(int x)`	`betrag: CPD    #0     ;x > 0?`
`{   return x > 0 ? x : -x;`	`        BGT    L0     ;if yes: return x`
	`        COMA          ;compute -x`
	`        COMB`
	`        ADDD   #1`
`}`	`L0:     RTS           ;return`
`void main(void)`	`main:   . . .`
`{   . . .`	
`    c16 = betrag(a16);`	`        LDD    a16    ;Pass parameter`
	`        JSR    betrag;Call subroutine`
`    . . .`	`        STD    c16    ;Store result`
`}`	

Simplest way to pass parameters: Parameters in register(s) here D

Return value(s) in register(s) here D

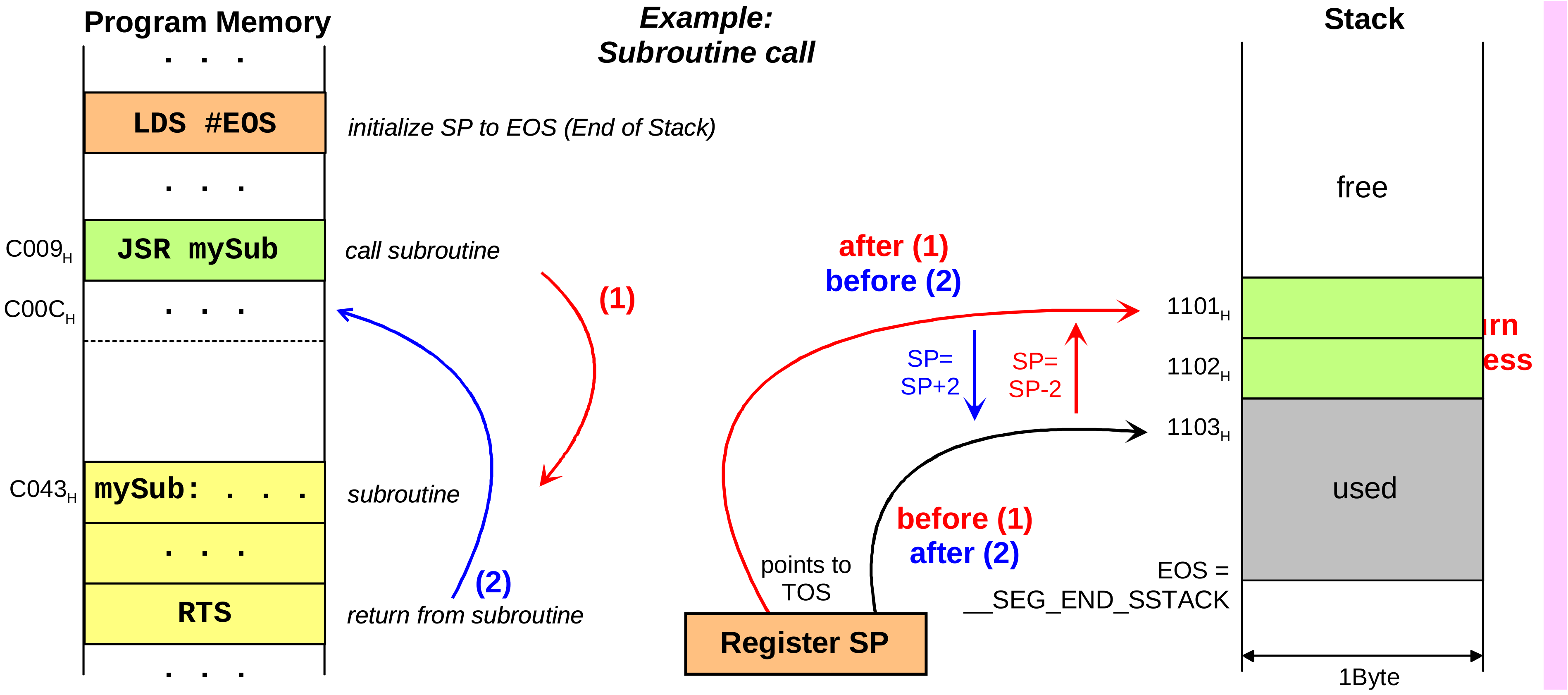
Functions with many parameters: Pass parameters via Stack, see chapter 4.

Note: Depending on configuration, the C-compiler may optimize the code and thus the assembler code generated will look different from the code shown here. For the example programs optimization was turned off.

2.8 Stack

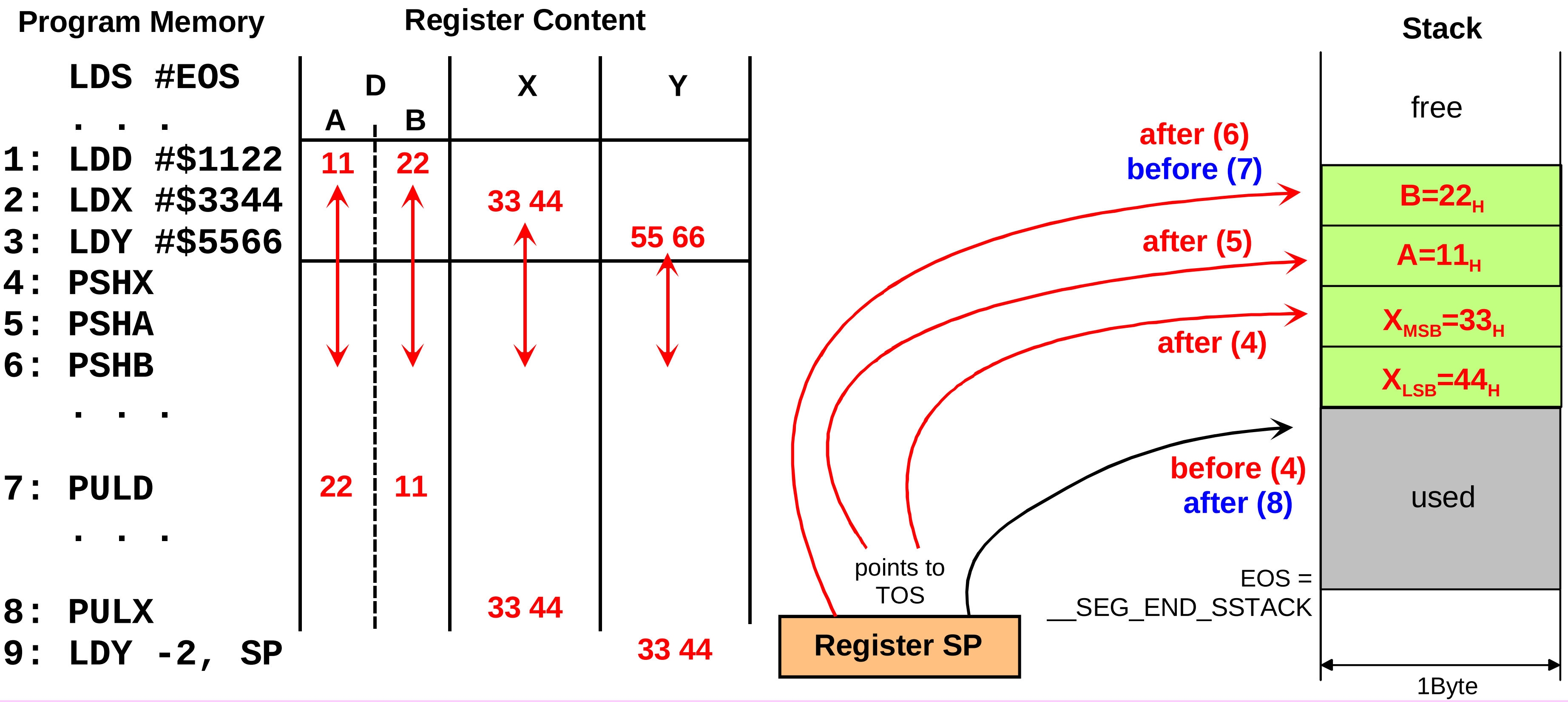
Purpose: RAM memory range to temporarily save registers and subroutine addresses

Idea: Last-In-First-Out-(LIFO) memory, filled from the end (End of Stack EOS)
Read/write access with register-indirect addressing via the stack pointer SP.
SP points to the latest byte stored on the stack (Top of Stack TOS)



2.8 Stack

Example: Save registers to Stack (CodeWarrior project AsmIntro.mcp)

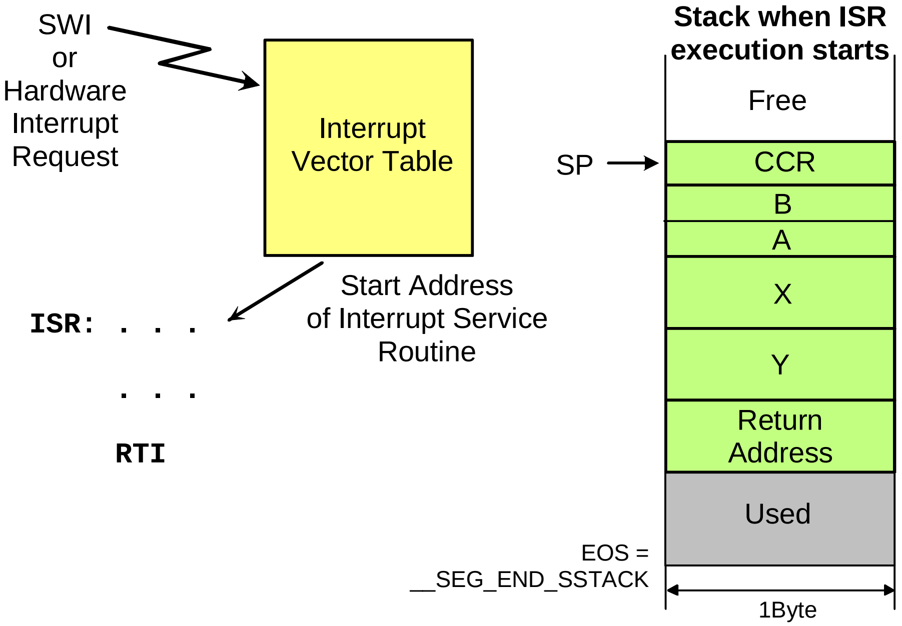


Program Memory	Register Content			
	D		X	Y
	A	B		
LDS #EOS				
. . .				
1: LDD #$1122	11	22		
2: LDX #$3344			33 44	
3: LDY #$5566				55 66
4: PSHX				
5: PSHA				
6: PSHB				
. . .				
7: PULD	22	11		
. . .				
8: PULX			33 44	
9: LDY -2, SP				33 44

- Allocate stack in RAM, automatically done by Linker [*1]
- Initialize stack pointer SP at begin of program: **`LDS #__SEG_END_SSTACK`** [*1]
- Stack pointer managed (increment/decrement) automatically by hardware
- Number of bytes stored on stack and retrieved from stack must be balanced in a program

2.8 Stack

- Interrupt-Service-Routines (see chapter 3) do automatically save and restore the register set on the stack:



*1 The CodeWarrior HCS12 development tools do define the stack size in linker control files **Simulator_Linker.prm** and **Monitor_Linker.prm.** The default size is **STACKSIZE 0x100** (256 byte). The linker provides a symbol **__SEG_END_SSTACK**, which points to the end of the stack. Assembler programs use this to initialize the stack pointer SP:

```
; Import symbols
XREF __SEG_END_SSTACK              ; End of stack

; Begin of program code
main:     LDS  #__SEG_END_SSTACK   ; Initialize stack pointer
```

2.9 Instruction Size and Execution Speed

- Instruction Size (Opcode length)

  HCS12 opcodes are 1 or 2 Byte long plus a variable number of bytes for a direct operand address or an immediate operand or an operand index. Constants are stored as 5, 9 or 11bit values when possible, to save memory space. Total instruction length is 1 to 6 byte.

- Execution Time (Instruction clock cycles)

  The number of clock cycles required to execute an instruction depends on the length of the instruction (cycles to read the instruction from memory), the location of the operands and result (read/write registers or memory) plus the actual execution of the operation.

  Reading/writing 2 bytes from a register or internal ROM/RAM memory typically takes 1 CPU clock cycle. See next page for examples.

- Detailed info can be found in literature reference [3.1, chapter 6.7 and appendix A], but is hard to read, because there are many dependencies. An easy way to find out is as follows:
  - The size of an instruction (including operands) can be seen in the Disassembly listing of the IDE’s source code editor (right click to open the listing) or in the Disassembly-Window of the debugger. The total size of a program can be found in the Linker/Locator’s Map file.
  - The execution time of an instruction or program can be “measured” in the HCS12 simulator (debugger in simulation mode), see CPU Cycle display in the debugger’s register window).

2.9 Instruction Size and Execution Speed

Rules of Thumb …

… for Instruction Size:	**Opcode + Operand Address Information**	
• Opcode length for most instructions:	1 byte	(MOVB, MOVW, TFR: 2 byte)
• Direct address operand:	2 byte	(if address ≥ 256)
• Immediate operand or index/offset constant:	1 or 2 byte	(if constant ≥ 256)
• Implicit register address:	0 (included in opcode)	

… for Execution Time:	**Fetch Instruction + Fetch Operand(s)**
	+ Execute Operation + Store Result
• Read/Write memory access:	1 cycle per 2 byte
(instruction or operand)	
• Register operand access:	0 (included in execute operation)
• Calculate pointers register-indirect:	1 cycle
memory-indirect:	2 cycles (includes read pointer from memory)
• Execute arithmetic/logic operation:	1 cycle

2.9 Instruction Size and Execution Speed

- Instruction size and speed depend on type and operand addressing mode. Examples:

Address mode [*1]		*Instruction*	*Length in byte*	*Speed in CPU-clock cycles* [*2]
Source operand	*Destination operand*			
Immediate (IMM)	Register	**`LDD #1234`**	3	2
Register indirect (IDX)	Register	**`LDD 0, X`**	2	3
Register indirect with increment	Register	**`LDD 2, X+`**	2	3
Memory direct (EXT)	Register	**`LDD var1`**	3	3
Register indirect with index (IDX2)	Register	**`LDD var1, X`**	4	4
Memory indirect with index ([IDX2])	Register	**`LDD [var1, X]`**	4	6
Register	Register	**`TFR D, X`**	2	1
Register indirect	Register indirect	**`MOVW 0, X, 0, Y`**	4	5
Memory direct	Memory direct	**`MOVW var1, var2`**	6	6
Direct		**`JMP address`**	3	3

Direct		**`JSR address`**
		`JSR [address]`
			3
			3
				4
				7

Implicit		**`RTS`**	1	5
Register implicit		**`INX`**	1	1
Memory direct	Register implicit	**`ADDD var1`** (16+16bit)	3	3
Register implicit	Register implicit	**`EMUL`** (16x16bit)	1	3
Register implicit	Register implicit	**`EDIV`** (32/16bit)	1	12

[*1] **`var1`**, **`var2`** … 16bit variables in internal ROM/RAM

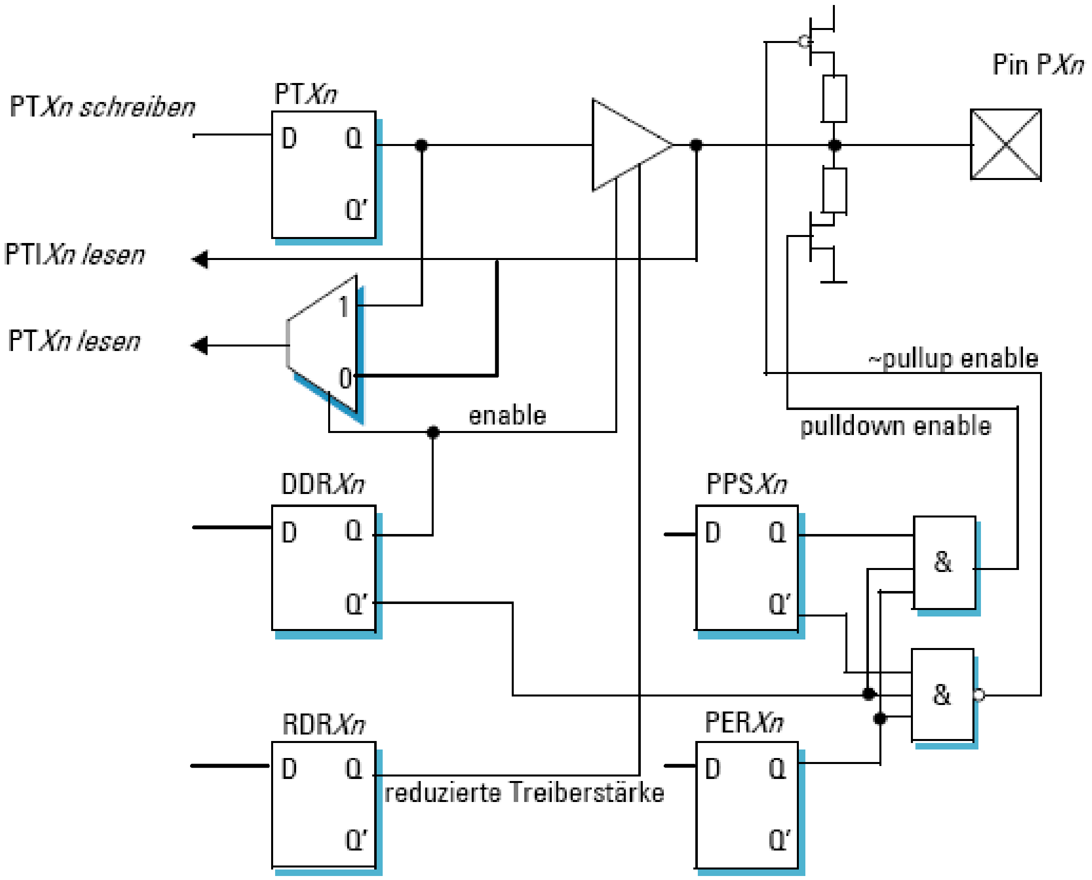
[*2] CPU clock period 42ns @ f_{BUSCLK}=24MHz

Chapter 3

Peripherals, Digital, Analog and Timed Input/Output, Interrupts

3.1 Digital Input and Output

(General Purpose Input/Output GPIO, see [3.3 and 3.10])



Circuit diagram of a single digital-input/output pin x.n (DDRx.n = 1 → Pin n of port X is configured as output, with n=0…7 and x=A, B, E, H, J, K, M, P, S, T)

- The CPU has several groups of 8 digital inputs/outputs (ports), which can be configured bitwise via the group's Data Direction Register **DDRx** as input or output.
- Reading and writing of port pins is done via data registers **PTx** or **PORTx** via MOVB instructions (8bit) or via BSET/BCLR to write and BRCLR/BRTST to test single pins.
- Via register **RDRx** (Reduced Driver) the driver signal strength of the port pin can be reduced, to improve the EMC properties.
- Via **PPSx** (Port Polarity Select) a pull-up or pull-down resistor can be selected and with **PERx** (Pull Enable Register) these resistors will be enabled.

3.1 Digital Input and Output

Most ports have alternative functions, e.g. ports A and B may be used as digital inputs/outputs *or* as external multiplex address/data bus. After Reset all ports are digital inputs. To use a port pin as output, configure it in the port's DDR register. If a port's alternative function is activated, the port (pin) can no longer be used as digital input/output:

Port x	*Data register/ Data direction*	*Notes/Restrictions*	*May trigger interrupt*	*Alternative Function*
A	PORTA / DDRA	Pull-Up resistors and driver strength only for complete port, not bitwise (uses register PUCR rather than PPSx and PERx, register RDRIV instead of RDRx)	No	Multiplex address/data bus
B	PORTB / DDRB			
E	PORTE / DDRE	Like PORTA; pins 0 and 1 only as inputs	No	Control signals for address data bus
K	PORTK / DDRK	Like PORTA		
H	PTH / DDRH		Yes	SPI interface
J	PTJ / DDRJ	Only 4 bit (J.0, 1, 6, 7) available	Yes	CAN or I^{2}C
M	PTM / DDRM		No	CAN interface
P	PTP / DDRP		Yes	PWM outputs SPI interface
S	PTS / DDRS		No	Serial interfaces SCI and SPI
T	PTT / DDRT		No	Timer input/output

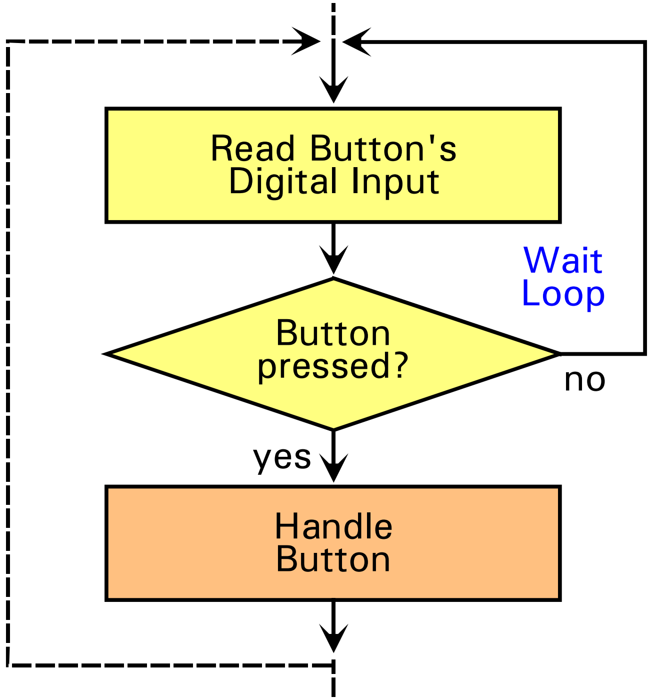
Ports H, J and P may generate edge-triggered interrupts (see chapter 3.2).

Ports are configured in the initialization phase of a program. Afterwards, ports are accessed via their data registers (see example **`BlinkingLED`** in chapter 2.2).

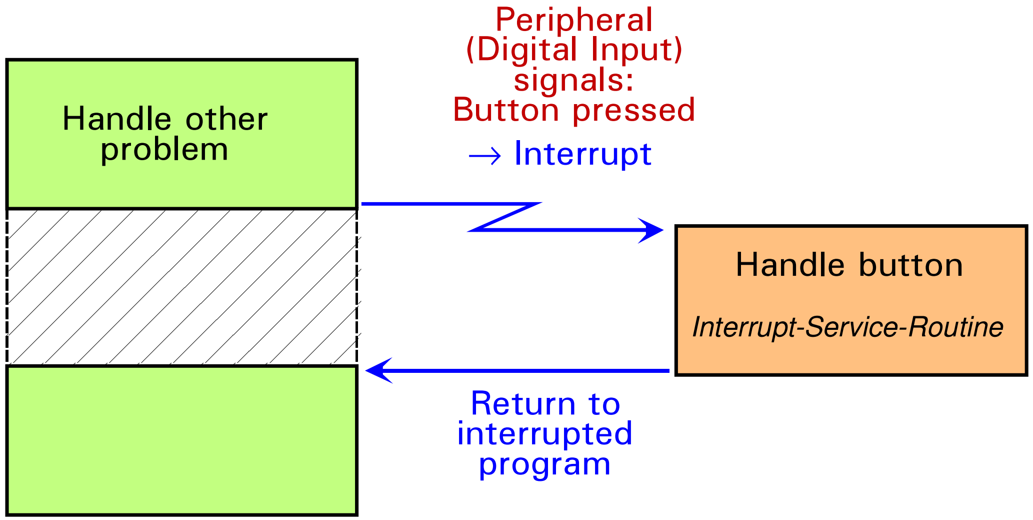
For port usage on Dragon12 boards see chapter 2.1 and [3.11].

3.2 Interrupts

Problem: Synchronize a program to an external event
e.g. a program, which shall react when a button is pressed

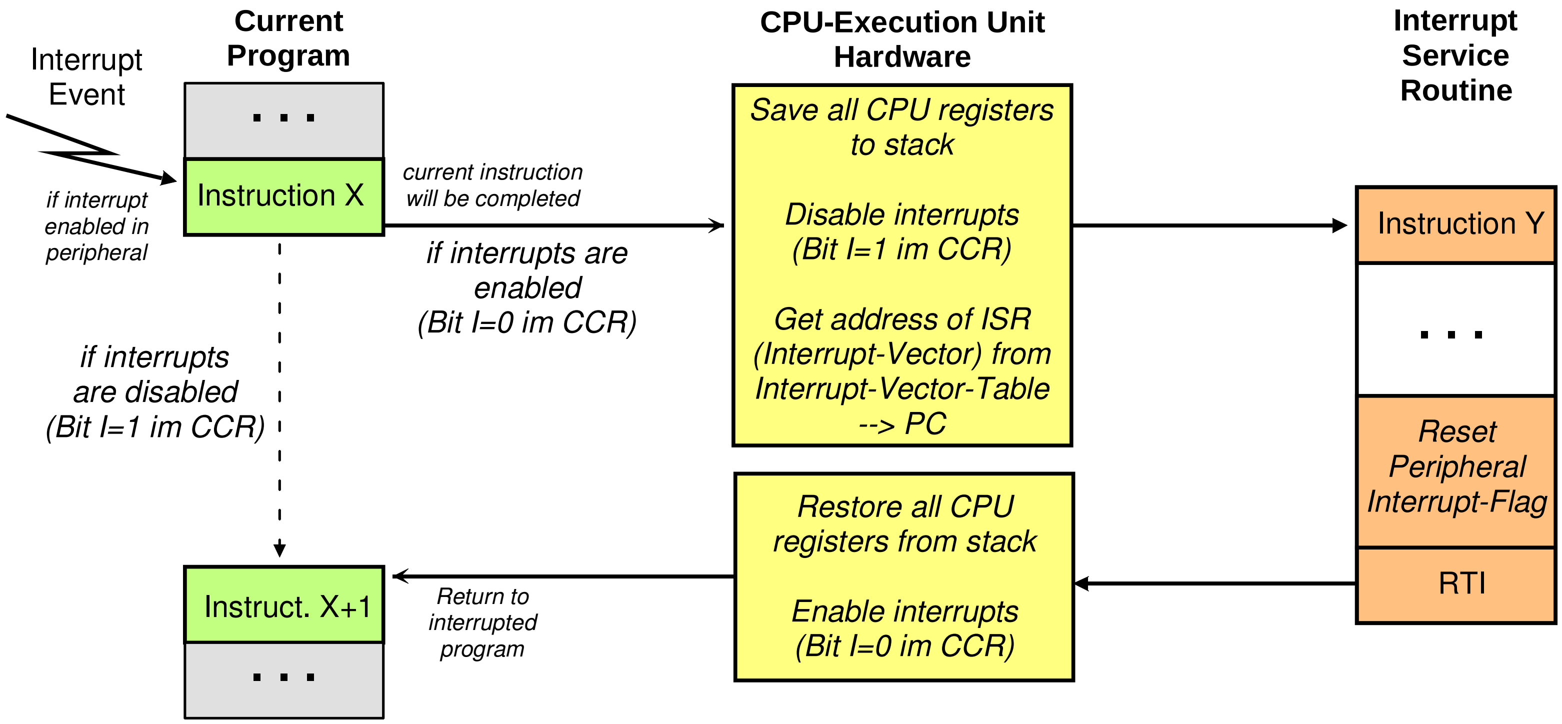


- Periodically reading the input signal (**Polling)** in a loop


Peripheral (Digital Input) signals: Button pressed
→ Interrupt
Handle other problem
Handle button
Interrupt-Service-Routine
Return to interrupted program

- Handling another problem (task) while waiting for the event
- Peripheral hardware signals the **Interrupt** to the CPU, when the button is pressed
- CPU interrupts the current task, handles the interrupt in an **Interrupt-Service-Routine ISR** and returns to the interrupted task again

Interrupt Handling Details:


Interrupt Event
Current Program
. . .
Instruction X
if interrupt enabled in peripheral
current instruction will be completed
if interrupts are enabled (Bit I=0 im CCR)
if interrupts are disabled (Bit I=1 im CCR)
Instruct. X+1
. . .
Return to interrupted program
CPU-Execution Unit Hardware
Save all CPU registers to stack
Disable interrupts (Bit I=1 im CCR)
Get address of ISR (Interrupt-Vector) from Interrupt-Vector-Table --> PC
Restore all CPU registers from stack
Enable interrupts (Bit I=0 im CCR)
Interrupt Service Routine
Instruction Y
. . .
Reset Peripheral Interrupt-Flag
RTI

Requirements for an interrupt based program:

- the interrupt mask bit I in CCR must have been cleared (global interrupt enable)
- the interrupt vector table (see next page) must have an entry of the ISR address
- the peripheral must be configured to trigger an interrupt (peripheral Interrupt Enable IE)
- the Interrupt Flag IF of the peripheral must be reset at the end of the ISR.

Interrupt-Vector-Table (selected interrupts only, for full table see [3.2])

No.	*Address*	*Usage*	*Maskable*
0	$FFFE	Reset	No *1
. . .	. . .	. . .	. . .
2	$FFFA	COP: Watchdog interrupt (Computer Operating not Properly)	Yes
3	$FFF8	TRAP: Unimplemented opcode	No *1
4	$FFF6	SWI: Software interrupt, called via instruction SWI	No *1
5	$FFF4	XIRQ: External, non-maskable interrupt request signal	No *2
6	$FFF2	IRQ: External interrupt request signal	Yes
7	$FFF0	**RTI: Real Time Interrupt**	Ja
. . .	. . .	. . .	. . .
8	$FFEE	**Timer channel 0**	Yes
. . .	. . .	. . .	. . .
15	$FFE0	**Timer channel 7**	Yes
. . .	. . .	. . .	. . .
20	$FFD6	SCI0: First serial interface	Yes
21	$FFD4	SCI1: Second serial interface	Yes
22	$FFD2	**ATD0: Analog To Digital converter**	Yes
. . .	. . .	. . .	. . .
25	$FFCC	**PTH: Digital input port H**	Yes
. . .	. . .	. . .	. . .
127	. . .	. . .	. . .

*1 „Non-maskable“ interrupts are independent from interrupt mask bit I in CCR.

*2 Input XIRQ can be disabled via mask bit X in CCR.

3.2 Interrupts

Interrupt Sources

- CPU reset triggered by an external signal, the watchdog or clock generator monitoring. These events use the interrupt-vector-table, but are no real interrupts, because they restart the CPU and do not return to the interrupted program.
- Hardware interrupts: Most peripherals can generate interrupts
- Software interrupt: Triggered by SWI instruction (used by debugger monitor program)
- Exceptions (traps): Triggered by some errors, e.g. trying to execute an undefined opcode.

Interrupt Priorities

- If several interrupt events do occur simultaneously, the ISRs will be executed in the order of their entries in the interrupt vector table with lowest number first, e.g. RTI before timer channel 0.
- Via register HPRIO the priority of one of the ISRs may be increased (not used in the lab). Other processors, e.g. x86, allow to change more priorities.
- If during an ISR other interrupt events shall be handled immediately, the interrupt mask can be cleared via the CLI instruction (nested-interrupts, not recommended)
- For each interrupt source, the latest interrupt event remains stored, till the interrupt flag in the associated peripheral is reset.

Communication between Interrupt Service Routines and other parts of a program

- Interrupt service routines are called by hardware asynchronously to other parts of a program. So an ISR does not have any call and return parameters (with the exception of SWI). If data exchange is required, global variables must be used.

Example in C:

(CodeWarrior project ButtonInterrupt.mcp)

- Trigger an interrupt by pressing button SW5 on the Dragon12 board (positive edge on CPU port H, bit 0)

```
void main(void)
{    EnableInterrupts;          //Allow interrupts
     . . .
     DDRB  = 0xFF;              //Port B.7...0 as outputs (LEDs)
     PORTB = 0x55;              //Turn on every second LED
     DDRH = 0x00;               //Configure port H.7…0 as inputs
     PPSH = 0x01;               //'1' trigger interrupt on positive slope on H.0
     PIEH = 0x01;               //'1' enables interrupt for input port H.0
     for(;;) { }                //Endless loop
}
interrupt 25 void ButtonISR(void) //ISR for interrupt 25 (port H interrupt)
{    PORTB = ~PORTB;            //Toggle LEDs on Port B
     PIFH = 0x01;               //’1’ resets the interrupt flag
}
```

Important registers for port H (see [3.3], chapter 3.3.5):

DDRH ... Data Direction Register
PTH ... Data register
PIEH ... Port Interrupt Enable register (1=enable, configured bitwise)
PPSH ... Port Polarity Select register (0=negative edge, 1=positive edge)
PIFH ... Port Interrupt Flag register (1=interrupt triggered, reset by writing a 1)

Note: In a practical application, buttons always must be debounced. This should be done by polling rather than interrupts.

3.2 Interrupts

Example in HCS12 Assembler:

(CodeWarrior project **ButtonInterruptAsm.mcp**)

- Requirements are the same as for the example in C

```
. . .
.vect:  SECTION             ; ROM: Interrupt vector section ----------------
        ORG   $FFCC
        DC.W  isr25         ; Interrupt vector for interrupt 25 (Port H)

.init:  SECTION             ; ROM: Code section ----------------------------
main:                       ; Begin of the program
Entry:  LDS  #__SEG_END_SSTACK  ; Initialize stack pointer
        CLI                 ; Enable interrupts

        . . .
        MOVB #$FF, DDRB     ; $FF -> DDRB:  Port B.7...0 as outputs (LEDs)
        MOVB #$55, PORTB    ; $55 -> PORTB: Turn on every other LED

        MOVB  #$00, DDRH    ; Configure port H.7...0 as inputs
        MOVB  #$01, PPSH    ; '1' trigger interrupt on positive slope
        MOVB  #$01, PIEH    ; '1' enables interrupt for input port H.0
loop:   BRA loop            ; Endless loop

; Interrupt Service Routine for interrupt 25
isr25:  COM  PORTB          ; Complement Port B: Toggle LEDs
        BSET PIFH, #1       ; Clear interrupt flag
        RTI                 ; Return from interrupt service routine (not RTS)
```

Note: In a practical application, buttons always must be debounced. This should be done by polling rather than interrupts.

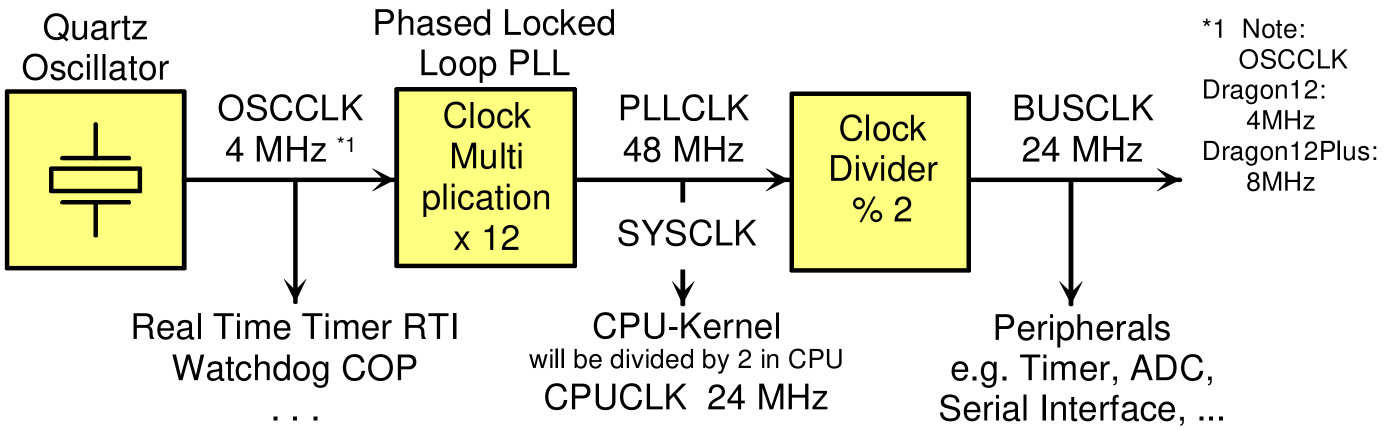
CPU State after Reset

- PC loaded with reset interrupt vector (points to Dragon12 board's debugger monitor program, which initializes the CPU's clock generator)
- I=X=S=1 in CCR register, i.e. interrupts masked (disabled), STOP instruction disabled
- Other registers, e.g. SP (!) undefined
- Peripherals turned off, i.e. the clock generator's PLL, the watchdog (COP) or the Real-time Timer (RTI)
- All digital input/output pins configured as inputs

Clock Generator CRG and periodical Real Time Interrupts RTI

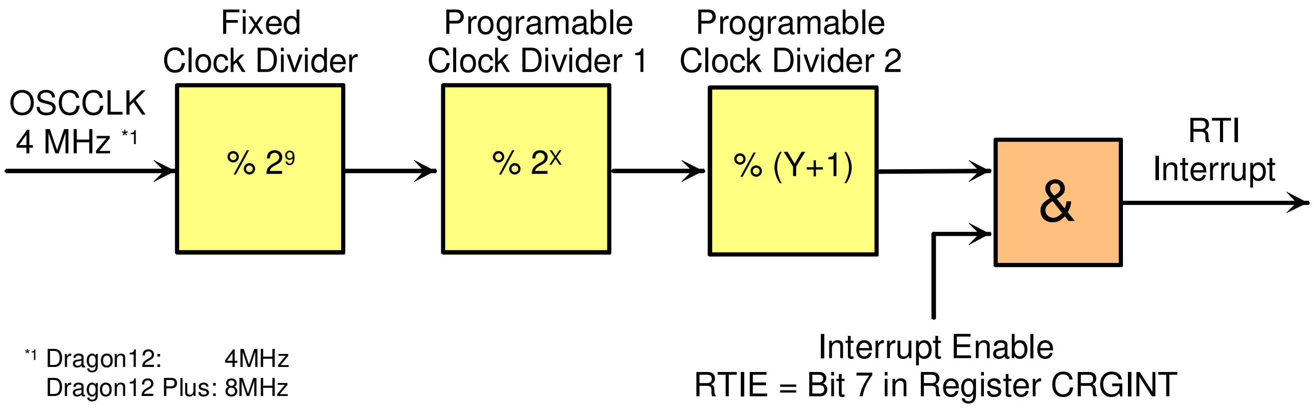
- **Clock Signals** (see Clock & Reset Generator Module CRG [3.4] and Appendix)

  The HCS12 has a complex programmable clock generator, which is configured by the debugger's monitoring program on the Dragon12 board as follows:



3.2 Interrupts

- **Real Time Interrupt RTI** (see Clock & Reset Generator Module CRG [3.4])
  Periodical interrupts can be generated using the Real Time Interrupt unit:



Fixed Clock Divider
Programable Clock Divider 1
Programable Clock Divider 2
OSCCLK
4 MHz *1
% 2⁹
% 2ˣ
% (Y+1)
&
RTI Interrupt
Interrupt Enable
RTIE = Bit 7 in Register CRGINT

*1 Dragon12: 4MHz
Dragon12 Plus: 8MHz

The interrupt frequency is defined via clock divider settings X and Y:

Bit	7	6	5	4	3	2	1	0
Register **RTICTL**	0	X (3bit)			Y (4bit)			

$$f_{RTI} = \frac{f_{OSCCLK}}{2^{9+X} \cdot (Y+1)}$$

Y=0…15, X=1…7 (Note: X = 000_B will disable the interrupt generation)

Register **CRGINT**: To enable the RTI interrupt (Real Time Interrupt Enable RTIE), set bit 7 in register CRGINT to 1. The other bits in this register must not be changed!

Register **CRGFLG**: At the end of the interrupt service routine the RTI Interrupt Flag RTIF, i.e. bit 7 in register CRGFLG, must be reset by writing a 1. The other bits in this register must not be changed!

Frequency range of the RTI interrupt

$$f_{RTI} = \frac{f_{OSCCLK}}{2^{9+X} \cdot (Y+1)}$$ with X=1…7, Y=0…15 and f_{OSCCLK} = 4 MHz

Maximum: smallest divider values X_{min} = 1 (not 0!), Y_{min} = 0 → RTICTL = 10_H

$$f_{RTI,\ max} = \frac{4\ \text{MHz}}{2^{9+1} \cdot (0+1)} = \frac{1}{256\mu s} \approx 3{,}9\ \text{kHz}$$

Minimum: biggest divider values X_{max}=7, Y_{max}=15 → RTICTL = $7F_H$

$$f_{RTI,\ min} = \frac{4\ \text{MHz}}{2^{9+7} \cdot (15+1)} = \frac{1}{262ms} \approx 3{,}8\ \text{Hz}$$

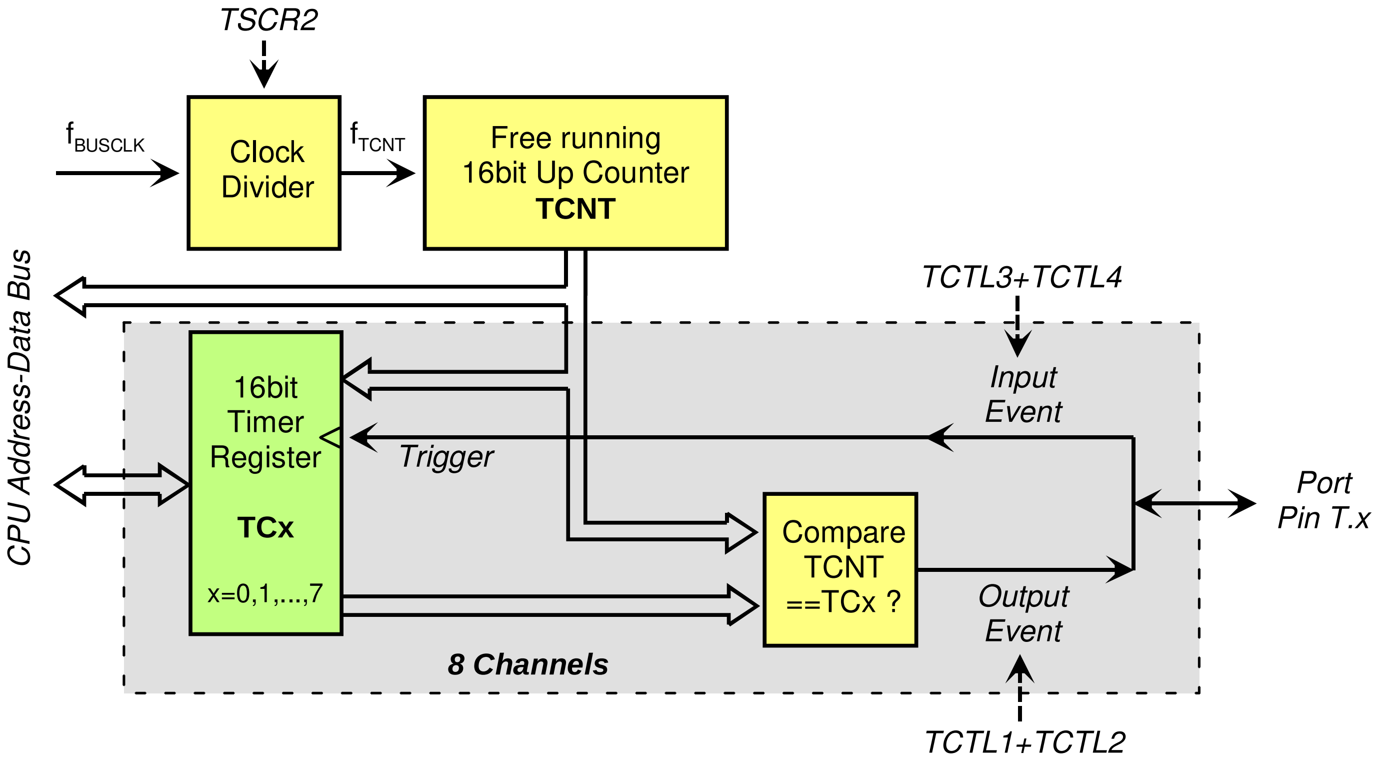
see also table on last page 3.61

3.3 Timer Unit

(see Enhanced Capture Timer ECT [3.5])

Every modern microcontroller has a powerful timer unit for tasks like

- measuring time differences/timeouts in programs
- measuring time instants of external events, pulse periods and pulse lengths of input signals (**Input Capture Mode**)
- generation of interrupts and output signals at programmable times (**Output Compare Mode**)



The HCS12 timer unit consists of

- a free-running 16bit counter TCNT with programmable clock frequency f_{TCNT}
- 8 channels, connected to Port T, which can be used as inputs in Input Capture Mode or as outputs in Output Compare Mode

3.3 Timer Unit

The Enhanced Capture Timer is a complex module with many configurable options. Only the most important options will be described here. The description assumes that the CPU was reset and only differences to the default settings after reset will be described. For more information see [3.5].

Configuration of the free-running 16bit Counter TCNT

- Control Registers

TSCR1 (8bit register)	Enable the timer unit	
	Bit 7 = 1	Enable
	Bit 6…0 = 0	Default after reset
TSCR2 (8bit register)	Set the timer clock frequency	
	Bit 7…3 = 0	Default after reset
	Bit 2…0	Clock divider x

Clock frequency $f_{TCNT} = \frac{1}{T_{TCNT}} = \frac{f_{BUSCLK}}{2^x}$

Dragon12 operates at f_{BUSCLK}=24MHz.

For x_{min}= 0 the 16bit counter has

- a clock period $T_{TCNT,min} = 1/f_{TCNT,max} =$ 42ns
- a counter period $T_{P,min} = 2^{16} \cdot T_{TCNT,min} =$ 2,7ms.

For x_{max}= 7 the 16bit counter has

- a clock period $T_{TCNT,max} = 1/f_{TCNT,min} =$ 5,3µs
- a counter period $T_{P,max} = 2^{16} \cdot T_{TCNT,max} =$ 350ms

The counter can be configured to generate an interrupt on counter overflow (not described here). In the associated ISR overflow events can be counted by software, so that the counter can be extended to more than 16bit.

Configuration of the 8 Timer Channels

- Control registers

TIOS (8bit register)	Select Input Capture or Output Compare Mode for each channel Bit y = 1 (y=0,1,…,7)	 Channel y in Output Compare Mode Default: y=0, i.e. Input Capture Mode
TIE (8bit register)	Interrupt Enable for each channel Bit y = 1 (y=0,1,…,7) Each channel has its own ISR, which will be called by its associated input or output event.	 Channel y does generate interrupts Default: y=0, i.e. no interrupt
TFLG1 (8bit register)	Interrupt Flag indicates an timer interrupt event Bit y = 1 (y=0,1,…,7)	 Channel y triggered an interrupt Must be reset in the ISR by writing a 1 to the channel's bit in TFLG1.

- Additional configuration registers for **Output Compare Mode**

	Bit 7	6	5	4	3	2	1	0
TCTL1 (8bit)	Channel 7		Channel 6		Channel 5		Channel 4	
TCTL2 (8bit)	Channel 3		Channel 2		Channel 1		Channel 0	

Set the type of output event:

00 … Output pin not used (timer used for interrupt generation only).

01 … Toggle output pin

10 … Clear output to 0

11 … Set output to 1

3.3 Timer Unit

- Additional configuration registers for **Input Capture Mode**

	Bit 7	6	5	4	3	2	1	0
TCTL3 (8bit)	Channel 7		Channel 6		Channel 5		Channel 4	
TCTL4 (8bit)	Channel 3		Channel 2		Channel 1		Channel 0	

Set the type of input trigger event:

00 … Input not used

01 … Positive edge

10 … Negative edge

11 … Pos. & neg. edges

- Time-stamp registers for the 8 channels
- Channel 0

TC0
(16bit register)

Software writes to this register in Output Compare Mode to set the "time" (counter value), when the channel's next output event shall be triggered.

. . .

. . .

- Channel 7

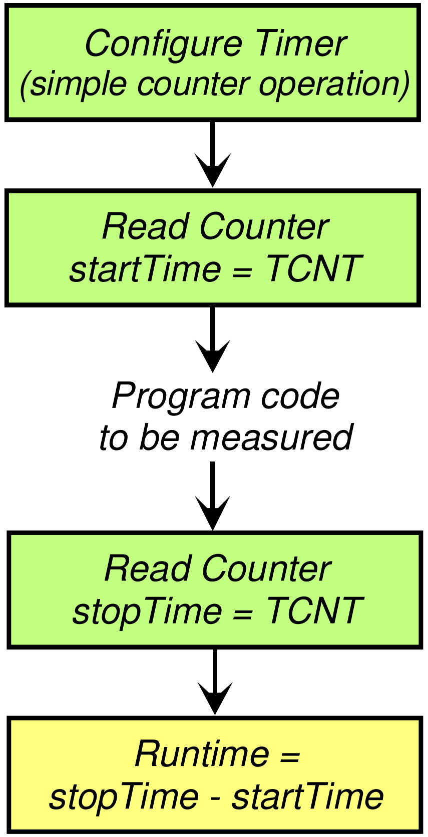
TC7
(16bit register)

Software reads this register in Input Capture Mode to get the "time" (counter value) of the channel's last input event.

As the timer clock frequency is high, the timer value TCNT changes fast and overflows periodically. Software therefore

- must read TCNT with a single 16bit instruction, e.g. MOVW (never use two sequential 8bit instructions MOVB, because the timer value may change before the second byte is read).
- When using TCNT to measure time, the time between events must never be greater than one counter period, i.e. $2^{16}/f_{TCNT}$. In this case, the CPU's mod 2^{16}-arithmetic will handle timer overflows automatically, otherwise overflows must be counted and handled by software.

Example: Program Code for Run-time Measurement (CodeWarrior project timer1C.mcp)



Configure Timer (simple counter operation)

Read Counter startTime = TCNT

Program code to be measured

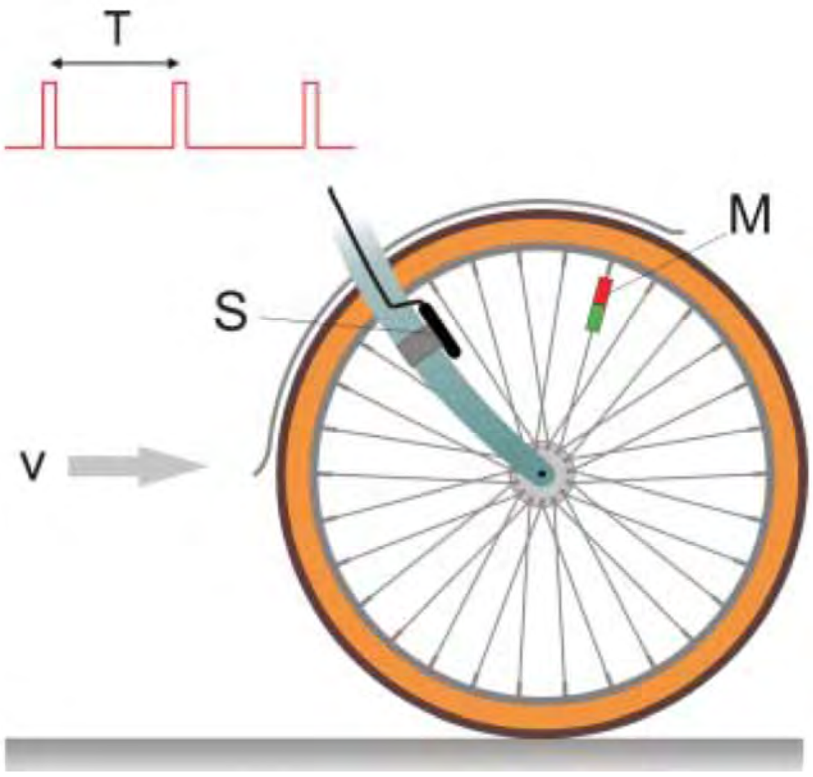
Read Counter stopTime = TCNT

Runtime = stopTime - startTime

```
// --- Global variables ------------------------------
unsigned int   startTime;    // TCNT at start
unsigned int   stopTime;     // TCNT at end
unsigned long deltaTime;     // stopTime-startTime

void main(void)
{ EnableInterrupts;          // Needed for debugger

//--- Initialize timer --------------------------------
   TSCR1 = TSCR1 | 0x80;     // Enable timer module
   TSCR2 = 0x07;             // Timer clock period
                                   2^7/24MHz=5.3µs

   startTime = TCNT;         // Save start time

//--- Program code to be measured ----------
     . . .

//--- Compute run time --------------------------------
   stopTime  = TCNT;                 // Save stop time
   deltaTime = stopTime - startTime; // Run time (in
                                       clock periods)
   deltaTime = deltaTime * 128/24;   // Convert to µs

}             → Assembler version see appendix A
```

Note: Resolution is 1 timer clock period, i.e. 5.3µs measurement range: 1 counter period, i.e. max. 350ms
A smaller clock period will increase measurement resolution, but will reduce the measurement range.

Example: Measure the Period of a Pulse Signal in **Input Capture Mode**



The speed of a bicycle is measured via a pulse signal:

- The pulse signal is generated via a coil S and a rotating magnet M.
- When the magnet passes the coil, a voltage impulse is induced.
- A comparator converts the analog voltage impulses into a digital pulse signal.
- Wheel speed n $\sim \dfrac{1}{\text{impuls period T}}$
- With the known wheel radius r the bicycle’s speed v = 2 $\pi \cdot r \cdot n$ can be calculated

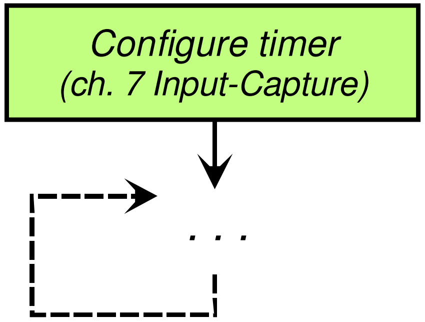
A similar arrangement with a tooth wheel and an Hall sensor, which generate ~ 60 pulses per revolution, is used to measure wheel speeds of cars in ABS/ESP systems.

3.3 Timer Unit

Code sample: (CodeWarrior project timer3C.mcp)

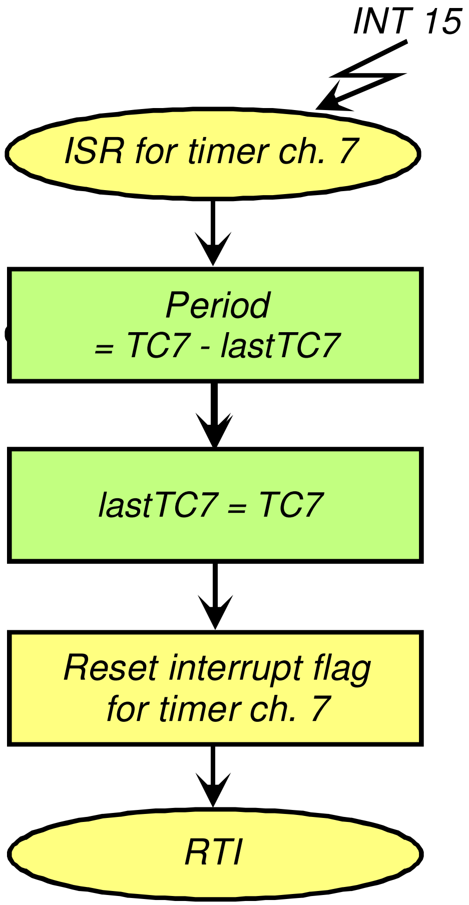
Pulse signal connected to port T.7 (used as input) with frequency range 10Hz … 10kHz

Main program:

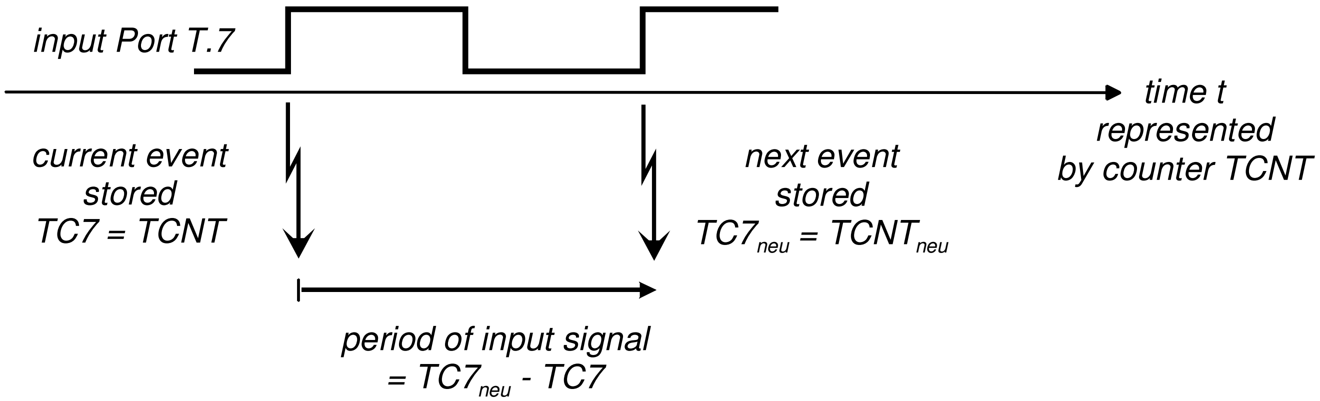

Configure timer
(ch. 7 Input-Capture)

. . .

```
//--- Global variables ---------------------
unsigned int signalPeriod = 0;  // Signal period in TCNT
                                          clock periods
unsigned int lastTC7 = 0;       // TC7 at last inp event

void main(void)
{ EnableInterrupts;      // Global interrupt enable
  . . .              (required, not only for debugger!)
//--- Initialize timer --------------------------------
  TSCR1 = TSCR1 | 0x80; // Enable timer module

  TSCR2 = 0x07;         // Timer clock period 2^7 / 24MHz
                        // = 5.3µs resol,clk period 350ms

  TIOS  = TIOS & ~0x80; // Timer ch. 7: input capture

  TCTL3 = 0x40;         // Trigger ch. 7 on rising edge

  TIE = TIE | 0x80;     // Enable interrupt for ch. 7


  for(;;) { }           // Infinite loop
}
```



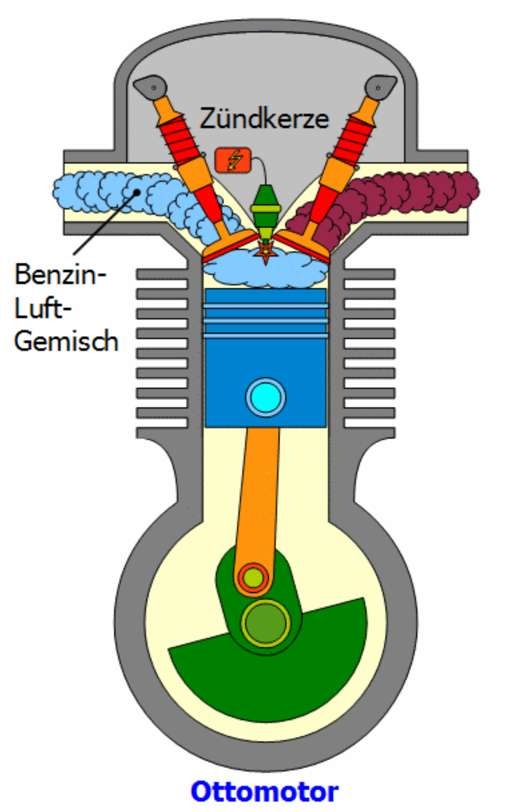
```
//--- Interrupt service routine for timer channel 7 ---
void interrupt 15 timer7Isr(void)
{
   unsigned int temp = TC7;      // Get current TC7

   signalPeriod = TC7 - lastTC7;// Compute signalPeriod
                                 (in timer clock periods)

   lastTC7 = temp;               // Save current TC7

   TFLG1 = TFLG1 | 0x80;    // Reset ch 7 interrupt flag
}
```

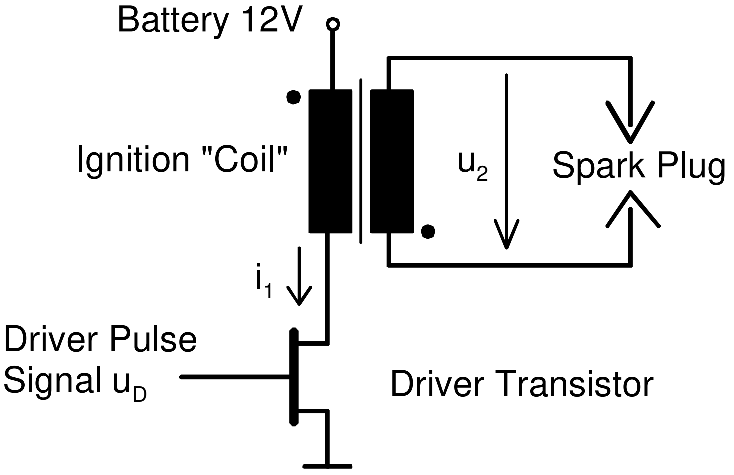


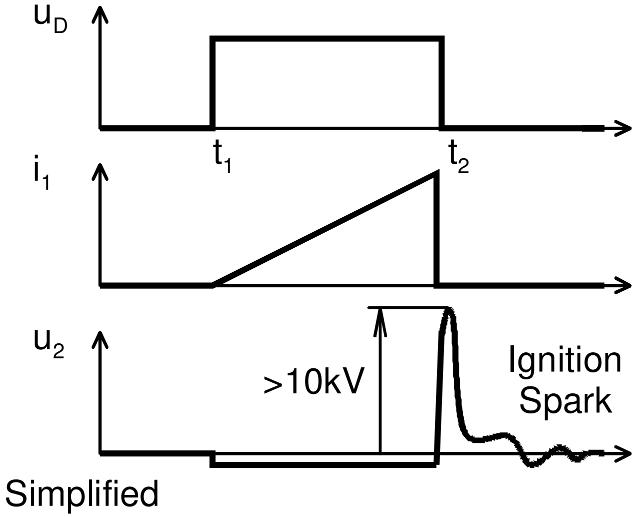
"time" as integer multiple of counter clock period TCNT

3.3 Timer Unit

Example:
Generate the driving pulse signal for the spark plugs of a combustion engine in **Output Compare Mode**







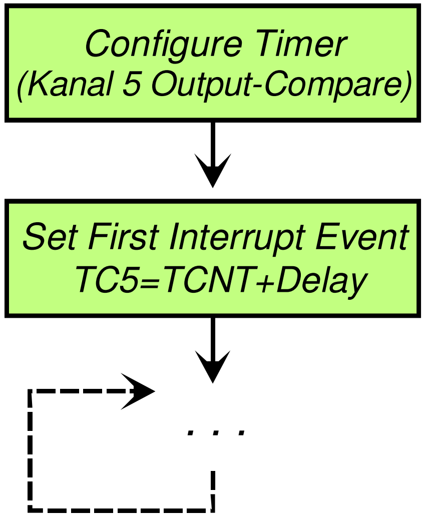
- The ignition spark is generated via the ignition “coil”.
- The ignition timing t_2 (but also t_1) must be controlled precisely, because they influence fuel efficiency and emissions → Timing errors < 1µs required.

3.3 Timer Unit

Code Example: (CodeWarrior project timer2C.mcp)
Driving the Dragon12 Beeper in Output-Compare Mode

A beeper (buzzer) on port T.5 (used as output) shall be driven by a 500Hz pulse signal with 1:1 duty cycle:

Main program:


Configure Timer
(Kanal 5 Output-Compare)

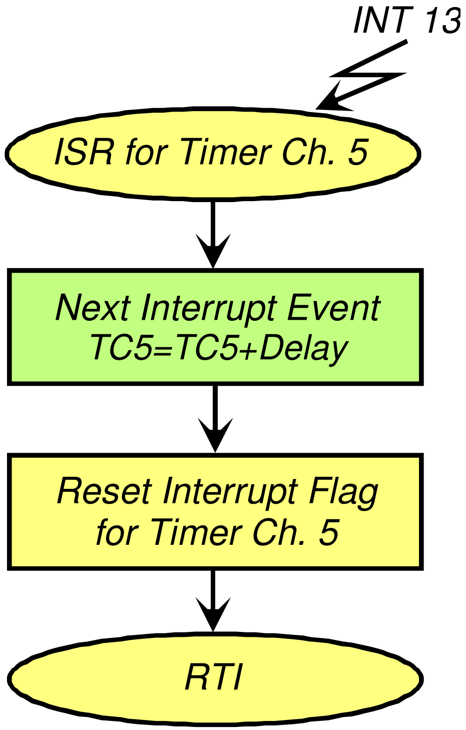
Set First Interrupt Event
TC5=TCNT+Delay

. . .

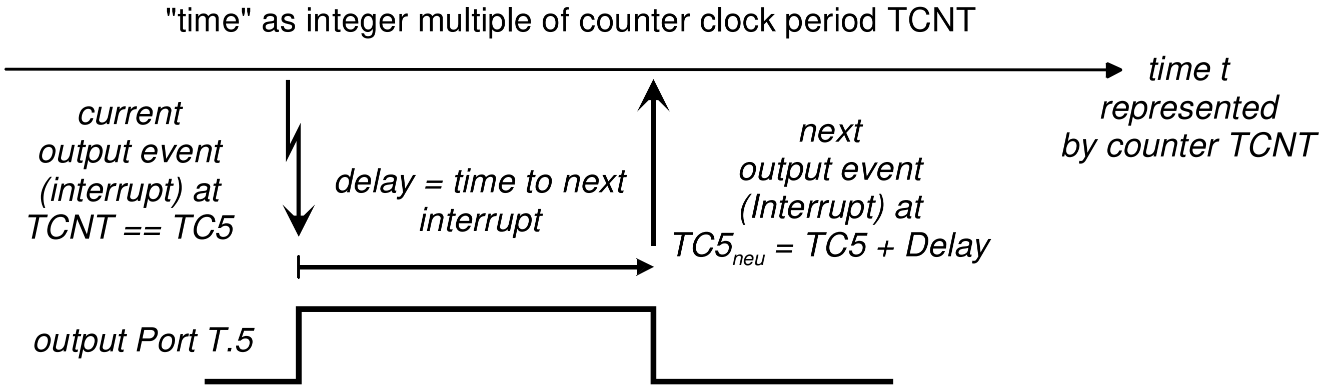
```
#define DELAY EQU (24000/128) // Delay 1ms * 24Mhz / 2^7

void main(void)
{ EnableInterrupts;     // Global interrupt enable
   . . .
//--- Initialize timer --------------------------------
  TSCR1 = TSCR1 | 0x80; // Enable timer module
  TSCR2 = 0x07;         // Timer clock period 2^7/ 24MHz

  TIOS  = TIOS | 0x20;  // Timer ch.5 as output compare
  TCTL1 = 0x04;         // On timer event toggle output
  TC5 = TCNT + DELAY;   // Set time of first event
  TIE = TIE | 0x20;     // Enable interrupt for ch. 5


  for(;;) { }           // Infinite loop
}
```

3.3 Timer Unit

ISR:

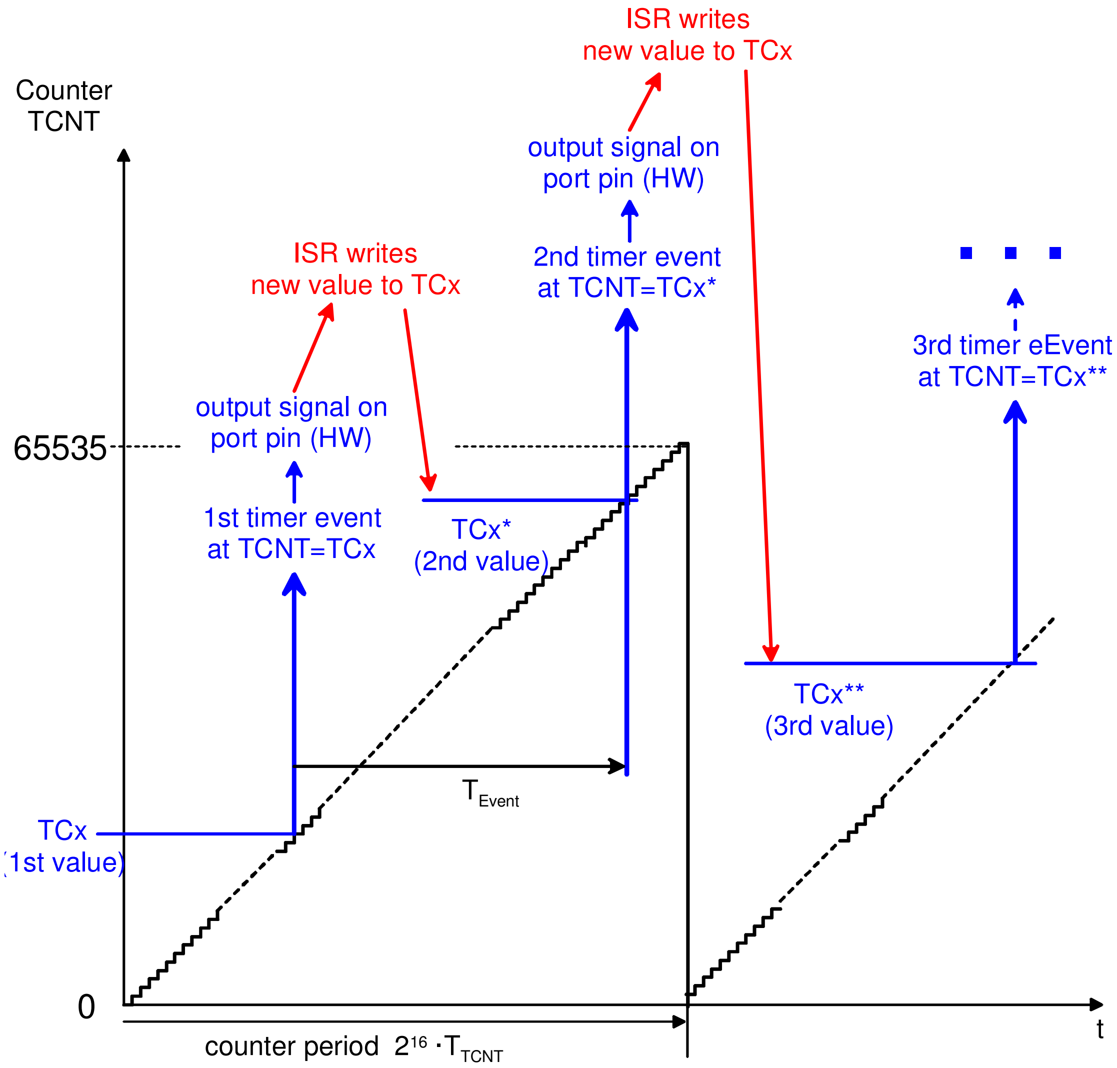


```
//--- Interrupt service routine for timer channel 5 ---
// Output port T.5 is toggled automatically, whenever
// this ISR is called

void interrupt 13 timer5Isr(void)
{
  TC5 = TC5 + DELAY;          // Set timer for next event
                          (Don't use TCNT here to reduce jitter)


  TFLG1 = TFLG1 | 0x20;     // Reset ch5 interrupt flag


}
```



By setting the bits in TCTL1 to 0, the output compare mode will trigger an ISR only at a pre-defined time, but does not change the port pins output signal.

Interaction between Hardware and Software in Output Compare Mode:



Minimum and maximum time between timer events:

$$T_{TCNT} << T_{ISR} < T_{Event} < 2^{16} \cdot T_{TCNT}$$

with T_{ISR} = duration of the ISR

Note:

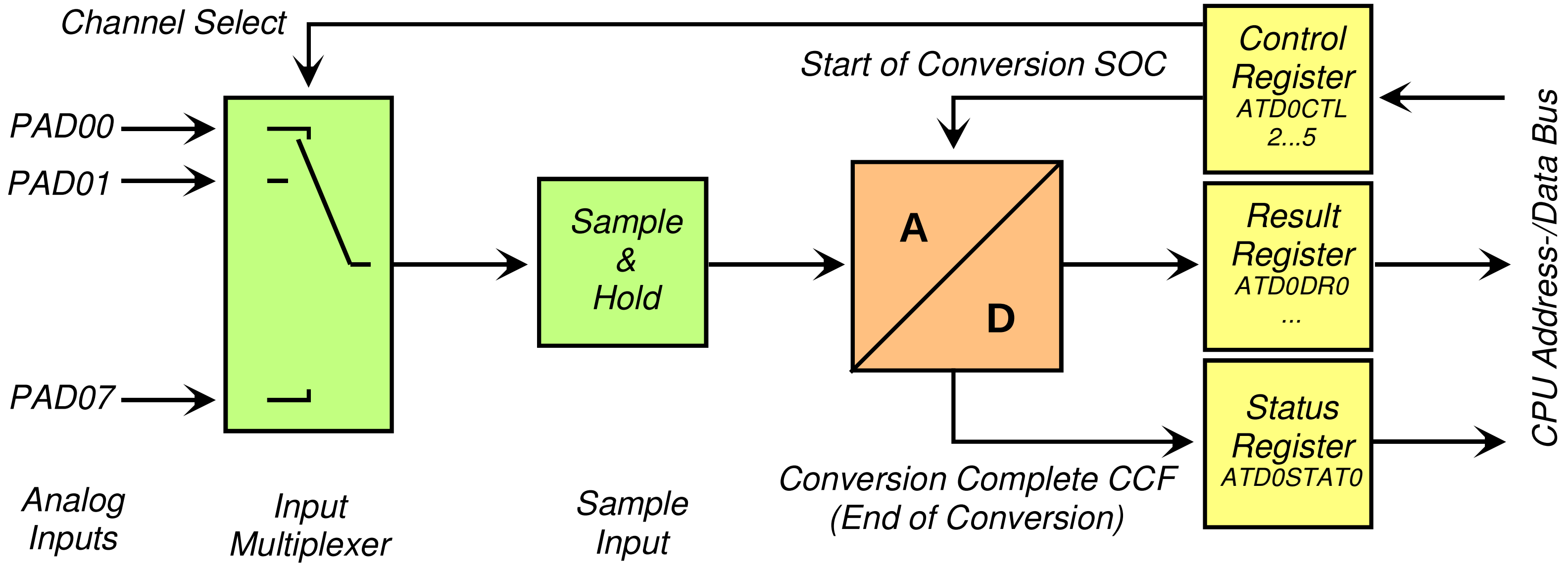
To generate times > 2^{16} T_{TCNT} , the timer can be configured to generate an interrupt, when the 16bit counter TCNT overflows:

- Timer Overflow Interrupt Enable:
  Register TSCR2 Bit 7 = 1
- Timer Overflow Interrupt Flag:
  Register TFLG2 Bit 7 = 1
- Interrupt Vector Table:
  Entry 16 @ address $FFDE

3.4 Analog to Digital Conversion

(see Analog To Digital Converter ATD [3.8])

- The HCS12 microcontroller of the Dragon12 board provides **two independent 10bit Analog-to-Digital Converters ATD0 and ATD1**.
- The converters use the **successive approximation** principle and have a **Sample & Hold** unit at their input.
- the **conversion time** is 14 clocks (2 clocks for switching the input multiplexer + 2 clocks for sampling + 10 clocks for the 10bit A/D-conversion) @ 2MHz clock, i.e. **7µs**.
- Via the **input multiplexer** each converter can select one of **8 analog input channels**: ATD0: Port PAD.07 … 00, ATD1: Port PAD.15…08.



3.4 Analog to Digital Conversion

Both converters ATD0 and ATD1 have the same set of registers and several operating modes. Here only the major options are discussed, for special modes see [3.8].

At program start a one-time configuration is done via the following control registers:

- Control Register

(all register 8bit, if not stated otherwise)

ATD0CTL2	Enable the ADC and interrupts	
	Bit 7 = 1	Enable the ADC module
	Bit 6 = 1	Automatic resetting of the CCF flag when reading the result registers
	Bit 5 … 2=0000_B	Miscellaneous options, don't change
	Bit 1 = 1	Enable interrupt after conversion completes
	Bit 0 = 1	Interrupt Flag, indicates an interrupt event, must be reset by writing a 1 into this bit
ATD0CTL3	Conversion sequence	
	Bit 7=0	Default
	Bit 6 … 3	Sequence Count SC, see below
	Bit 2 … 0 = 000_B	Default
ATD0CTL4	Resolution and conversion speed	
	Bit 7 = 0	10bit resolution (Bit 7 = 1 … 8bit)
	Bit 6,5=00_B	Sampling length 2 clocks (don't change)
	Bit 4..0=00101_B	Clock divider f_{ADC}=2MHz @ f_{BUSCLK}=24MHz (maximum clock frequency)
ATD0CTL5	Data format and start of conversion	
	Bit 7…5 = 100_B	Result in result register right-adjusted and unsigned, i.e. 0V = 0_D, 5V = 1023_D Conversion start triggered by software
	Bit 4	Multichannel conversion MULT, see below
	Bit 3 = 0	Default
	Bit 2 … 0	Channel select code C, see below

3.4 Analog to Digital Conversion

If the conversion is complete and the result available, can be polled via

• Status Register	**ATD0STAT0**	Bit 7 = 1	End of conversion EOC
		Bit 6 … 0	Other status info, not described here

The conversion result can be read from the 16bit **result register ATD0DR0** (and in registers ATD0DR1, …, ATD0DR7, see below).

Operating modes and start of a conversion:

A. Single measurement of a single channel

Required setting	in ATD0CTL3$_{6…3}$:	SC=0001$_B$ single measurement
	in ATD0CTL5$_4$:	MULT=0 no multi-channel operation
	in ATD0CTL5$_{2..0}$:	C = 0, 1, …, 7 select channel to measure
Start of conversion	when writing ATD0CTL5	
End of conversion	ATD0STAT0$_7$=1	Poll status register or configure interrupt
Conversion result	in ATD0DR0 (always, no matter which channel was selected)	

B. Multiple measurements of a single channel

Settings see A., except	in ATD0CTL3$_{6…3}$:	SC= 1, 2, …, 8 number of measurements
Conversion result:	in ATD0DR0, ATD0DR1, … (first result, second result, …)	

C. Single measurement of multiple channels (channels in ascending order)

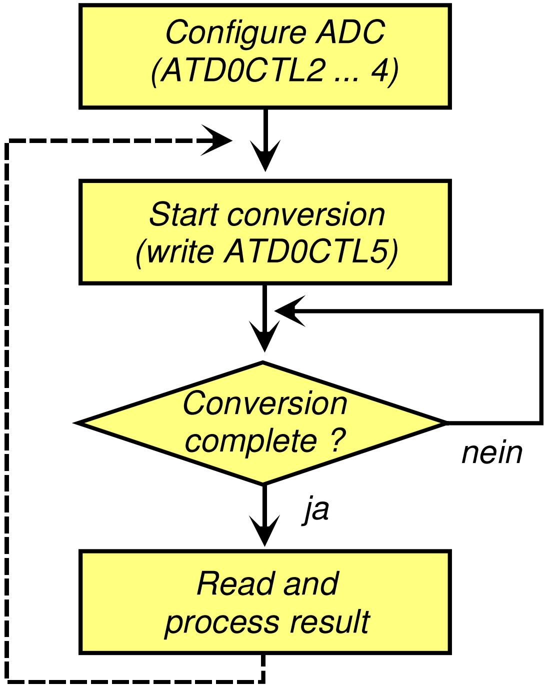
Settings see A., except in ATD0CTL3$_{6…3}$: SC= 1, 2, …, 8 number of channels

in ATD0CTL5$_4$: MULT=1 multi-channel mode

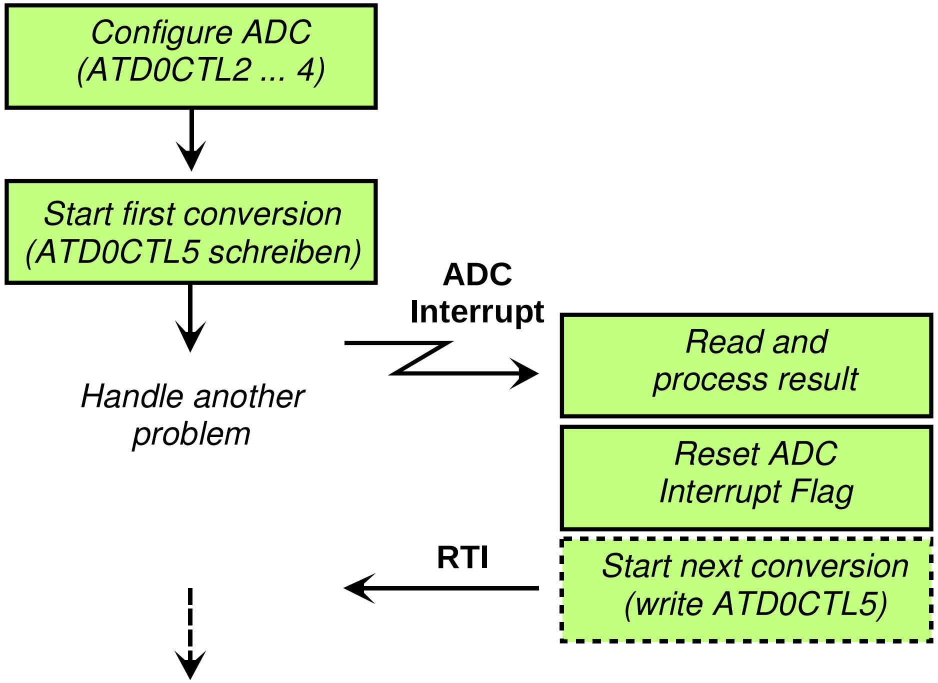
in ATD0CTL5$_{2…0}$: C = 0, 1, …, 7 first channel to be measured

E.g.: when starting at C=6 with SC=4, channels 6, 7, 0 and 1 will be measured.

Conversion result: in ATD0DR0, ATD0DR1, … (first channel, second channel, …)


Configure ADC (ATD0CTL2 ... 4)
Start conversion (write ATD0CTL5)
Conversion complete ?
nein
ja
Read and process result

Polling Mode


Configure ADC (ATD0CTL2 ... 4)
Start first conversion (ATD0CTL5 schreiben)
Handle another problem
ADC Interrupt
Read and process result
Reset ADC Interrupt Flag
Start next conversion (write ATD0CTL5)
RTI

Interrupt Mode

Example: Single Polling of Channel 7

```
main:   . . .

        MOVB #$C0, ATD0CTL2              ; Enable ATD, no interrupt

        MOVB #$08, ATD0CTL3              ; Single conversion only

        MOVB #$05, ATD0CTL4              ; 10bit, 2MHz ATD0 clock

        MOVB #$87, ATD0CTL5              ; Start conversion on channel 7

wait1:  BRCLR ATD0STAT0, #$80, wait1     ; Wait for End of Conversion (EOC)

        LDD  ATD0DR0                     ; Read conversion result → register D

        . . .
```

3.4 Analog to Digital Conversion

Example: (CodeWarrior project `ADCInterruptC.mcp`)

- Measure the analog signal on channel 7 (connected to a poti on Dragon12 boards)
- Output the arithmetic mean of 2 measurement values (in binary) to the LEDs on port B

```
. . .
//--- Global variables ---------------------
unsigned int value;          // Measurement value


void main(void)
{   EnableInterrupts;        // Enable interrupts

//--- Initialize ATD0 ----------------------------------------
  ATD0CTL2 = 0b11000010;     // Enable ATD0, enable interrupt

  ATD0CTL3 = 0b00100000;     // Sequence: 4 measurements

  ATD0CTL4 = 0b00000101;     // 10bit, 2MHz ATD0 clock

  ATD0CTL5 = 0b10000111;     // Start first measurement on single channel 7

  for(;;)                    // Infinite loop
  {
     // Show upper 8bits of the 10bit measurement value on LEDs 7...0
     PORTB = value >> 2;
  }
}
```

```
// --- ADC interrupt service routine ------------------------

void interrupt 22 adcISR(void)
{
   // Read the result registers and compute average of 4 measurements
   value = (ATD0DR0 + ATD0DR1 + ATD0DR2 + ATD0DR3) >> 2;

   ATD0CTL2 = ATD0CTL2 | 0x01;  // Reset interrupt flag (bit 0 in ATD0CTL2)

   ATD0CTL5 = 0b10000111;       // Start next measurement on single channel 7




}
```

Note: Result registers ATD0DR0, … must be **read** (to automatically reset the CCF flag), **before** starting the **next conversion.**

Instead of starting the A/D conversion via software (by writing to ATD0CTL5), the HCS12 allows other modes to trigger a conversion:

- Automatic Triggering: In this mode (scan mode or free running mode, activated via bit 5=1 in register ATD0CTL5) software only triggers the first conversion. All following conversions will start automatically after end of the previous conversion. Result registers can be read any time and always provide the latest result. However, in this mode sampling is asynchronous, i.e. the user program has no control of the signal sampling period.
- Hardware Triggering: Rather than by software, a conversion will be started by an external hardware signal (a rising or falling edge of an pulse signal at channel 7 of the ADC).

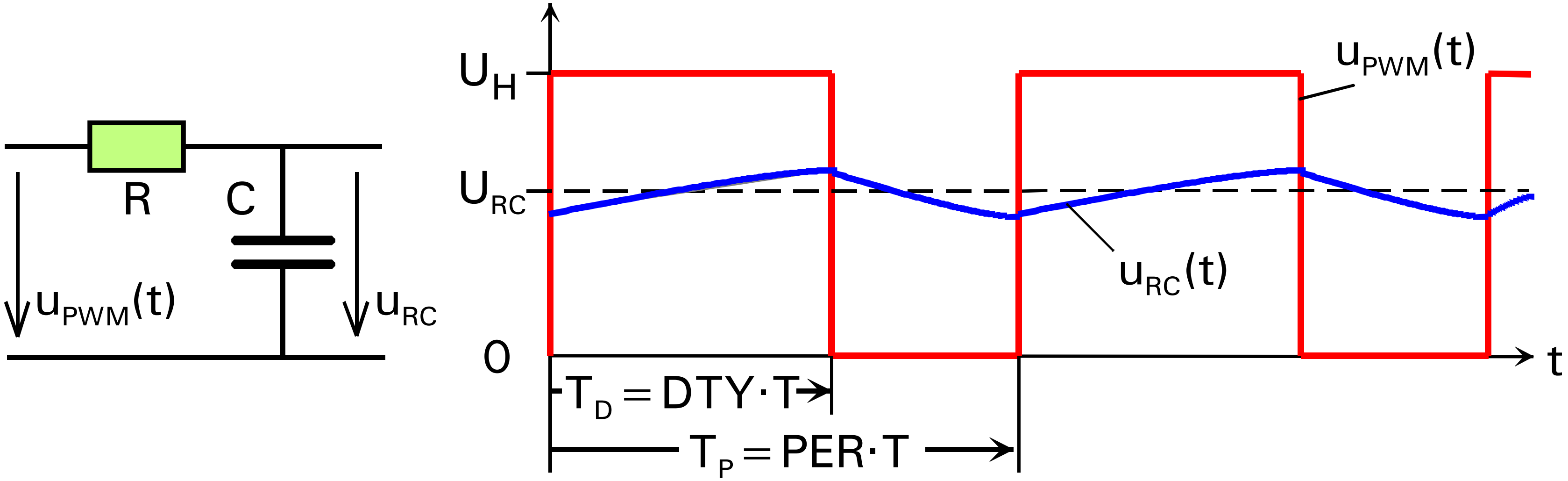
3.5 PWM Outputs

(see Pulse-Width-Modulation Unit PWM [3.7])

Purpose:

- Generation of pulse signals with programmable period T_P or pulse length(width) T_D

(D … Duty Cycle)



- Pseudo digital-to-analog conversion: Arithmetic mean $U_{RC} \approx U_{PWM} = U_H \frac{T_D}{T_P}$ for $RC >> T_P$

(valid only if PWMPOL=1)

- After filtering the output via a RC or RL low pass filter (e.g. the coil of a DC motor or solenoid) → control of servo motors, solenoids, lamps, …
- By changing T_D and/or T_P during run-time, the „analog" signal can be modulated (Pulse Width or Pulse Frequency Modulation)

The HCS12 PWM module has 8 PWM output channels (Port P.7…0). T_D and T_P can be set individually for each channel with a resolution of 8 bit:

- Register for Ch. x (all registers 8bit) (x=0,…,7)

PWMPERx	Period T_P as multiple of the clock period T_x (for maximum resolution use PWMPER=255)
PWMDTYx	Length of phase T_D as multiple of the clock period T_x (Make sure PWMDTYx < PWMPERx)
PWMCNTx	Counter register of PWM channel x (Clear to 0 to restart the PWM signal. Automatically done by reset, so normally not required).

There are three 8bit control registers with one bit per channel:

- Common Control Registers (1bit per channel)

PWME	Enable channel: Set a channel’s bit to 1, to start its PWM signal. Otherwise, the port pin can be used as normal digital input/output. Enable only, after all PWM channels including clock dividers have been configured.
PWMPOL	PWM signal polarity: When a channel’s bit is set to 1, the PWM signal period starts with the H phase, L otherwise.

3.5 PWM Outputs

PWMCLK

Selection of one of the clock signals $T_{A/B}$ or $T_{SA/SB}$:
If a channel's bit is set to 0, the channel uses the fast clock $T_{A\,/\,B}$, set to 1 for the slow clock $T_{SA\,/\,SB}$

The PWM module has a total of 4 clock signals, paired in two groups. For each channel one out of the two signals in a group can be selected via PWMCLK (see diagram on next page):

T_A or T_{SA} for PWM channels 0, 1, 4, 5 T_B or T_{SB} for PWM channels 2, 3, 6, 7

These clock signals are generated via clock dividers from the CPU's clock BUSCLK. The clock signals are programmed via clock dividers x_A or x_B and y_{SA} or y_{SB}:

$$T_A = 2^{x_A} \cdot T_{BUSCLK} \qquad T_B = 2^{x_B} \cdot T_{BUSCLK}$$

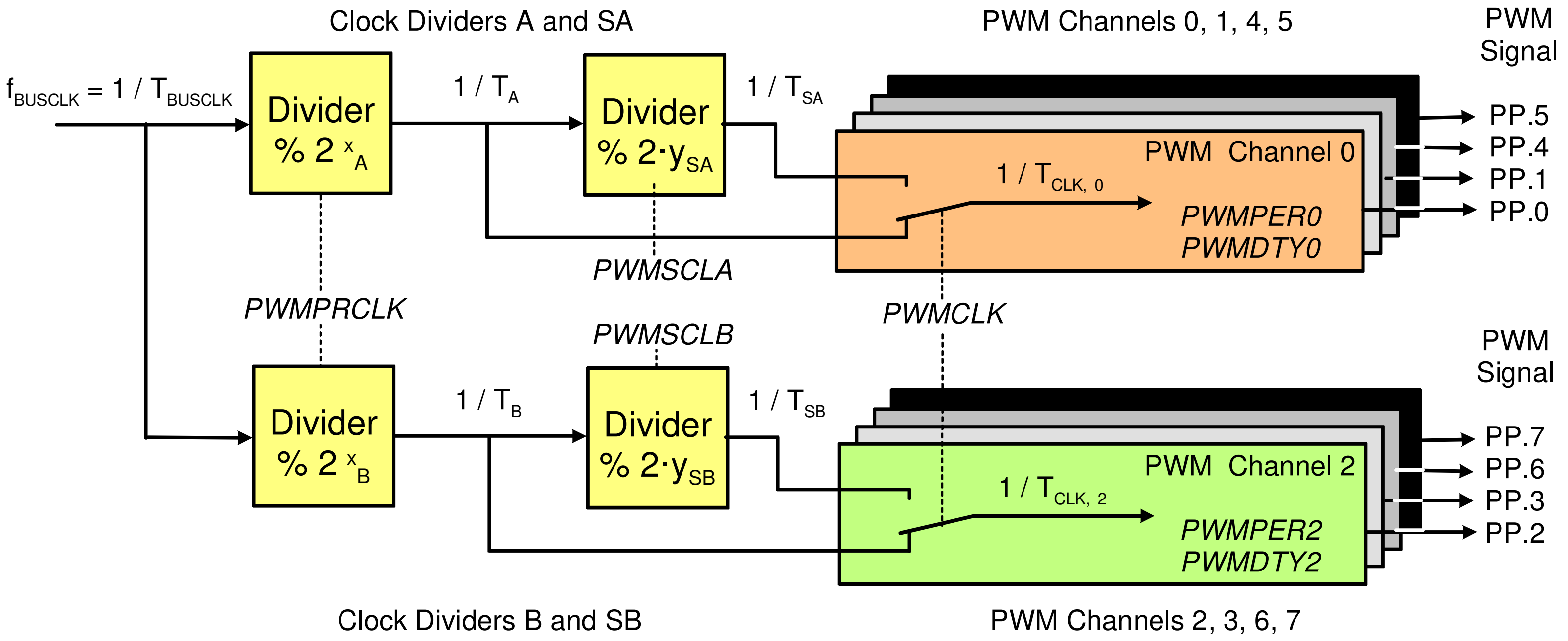
$$T_{SA} = 2 \cdot y_{SA} \cdot T_A \qquad T_{SB} = 2 \cdot y_{SB} \cdot T_B$$

Dragon12 uses $T_{BUSCLK}=1/f_{BUSCLK}=1/24MHz$. The clock dividers x_A, x_B, y_{SA} and y_{SB} are set via the following registers:

Bit	7	6	5	4	3	2	1	0	
Register **PWMPRCLK**	0	x_B (3bit)			0	x_A (3bit)			x_A, x_B = 0, 1,…, 7
Register **PWMSCLA**	y_{SA} (8bit)								y_{SA}, y_{SB} = 1, 2,…, 256
Register **PWMSCLB**	y_{SB} (8bit)								y=0 is interpreted as y=256

The PWM module has several options and operation modes, which are not discussed here. These options are turned off automatically after reset.

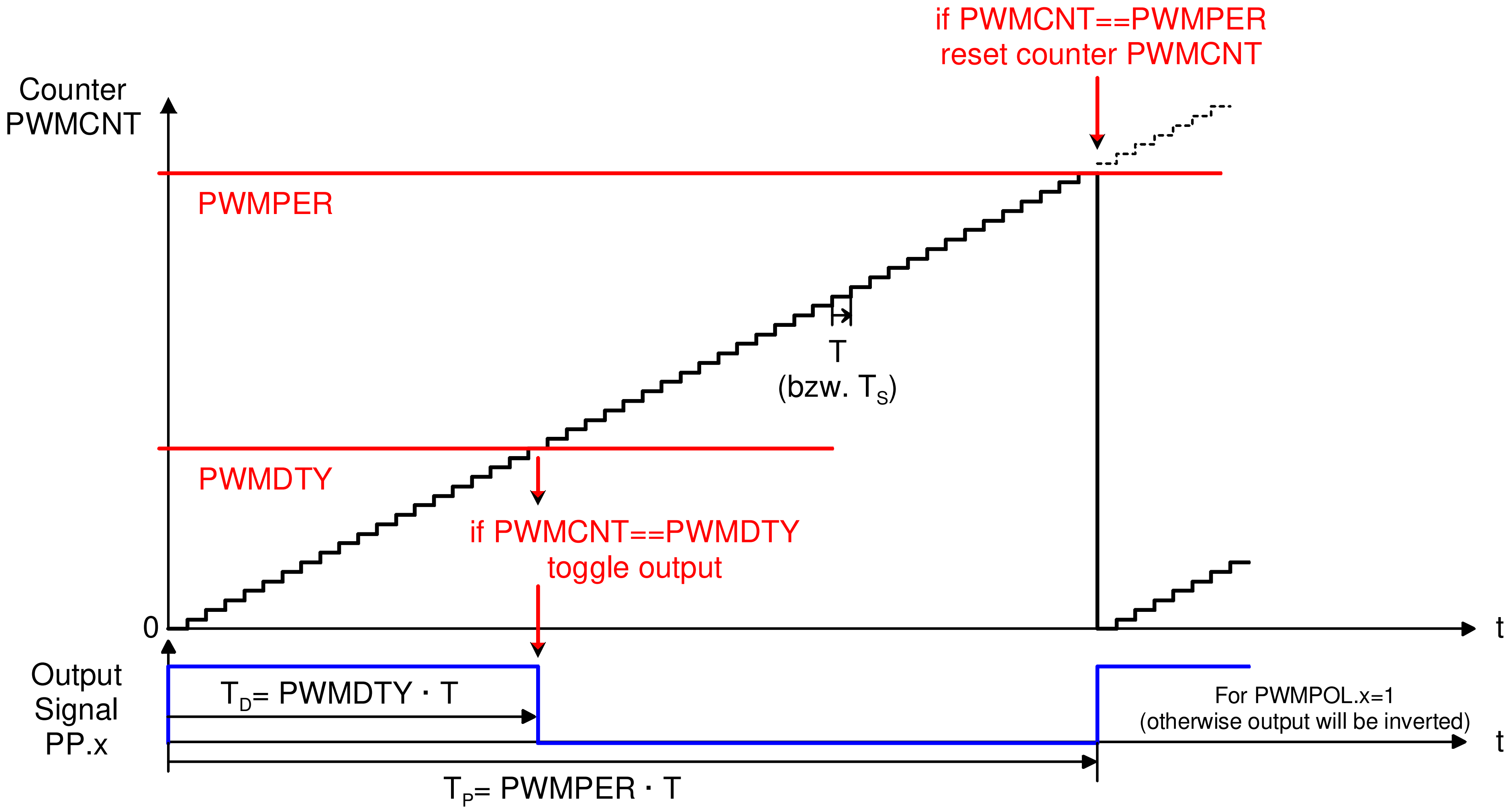
Structure of the PWM Module



- 2 clock divider groups μ=A, B with 2 clocks each: $T_\mu = 2^{x_\mu} \cdot T_{BUSCLK}$ and $T_{S\mu} = 2 \cdot y_{S\mu} \cdot T_\mu$
- Channels $\nu = 0, 1, 4, 5$ use clock $T_{CLK\nu} = T_A$ or T_{SA} dependent on the setting of PWMCLK.ν
- Channels $\nu = 2, 3, 6, 7$ use clock $T_{CLK\nu} = T_B$ or T_{SB} dependent on the setting of PWMCLK.ν
- Period of the pulse signal of channel ν is $T_{P\nu} = PWMPER_\nu \cdot T_{CLK\nu}$
- Length of the H-phase (if PWMPOL.ν=1) or the L-phase (if PWMPOL.ν=0) for channel ν

  $T_{D\nu} = PWMDTY_\nu \cdot T_{CLK\nu}$

Operation of a PWM channel



PWM Signal Frequency Range

What are the minimum and the maximum frequencies of a PWM signal on a Dragon12 board? The duty cycle T_D / T_P shall be configured for full 8bit resolution.

- 8bit duty cycle resolution requires: PWMPER = 255

- Maximum PWM frequency, if $T_{A/B}$ is used with $x_{A/B}$= 0

  $T_{Pmin} = 255 \cdot T_{A/Bmin} = 255 \cdot 2^0 \cdot 1/24MHz = 11µs \rightarrow f_{Pmax} = 1/T_{pmin} = 94kHz$

- Minimum PWM frequency, if $T_{SA/SB}$ is used with $x_{A/B}$=7 and $y_{SA/SB}$=256

  $T_{Pmax} = 255 \cdot T_{SA/SBmax} = 255 \cdot 2^7 \cdot 2 \cdot 256 \cdot 1/24MHz = 700ms \rightarrow f_{Pmin} = 1/T_{Pmax} = 1{,}4Hz$

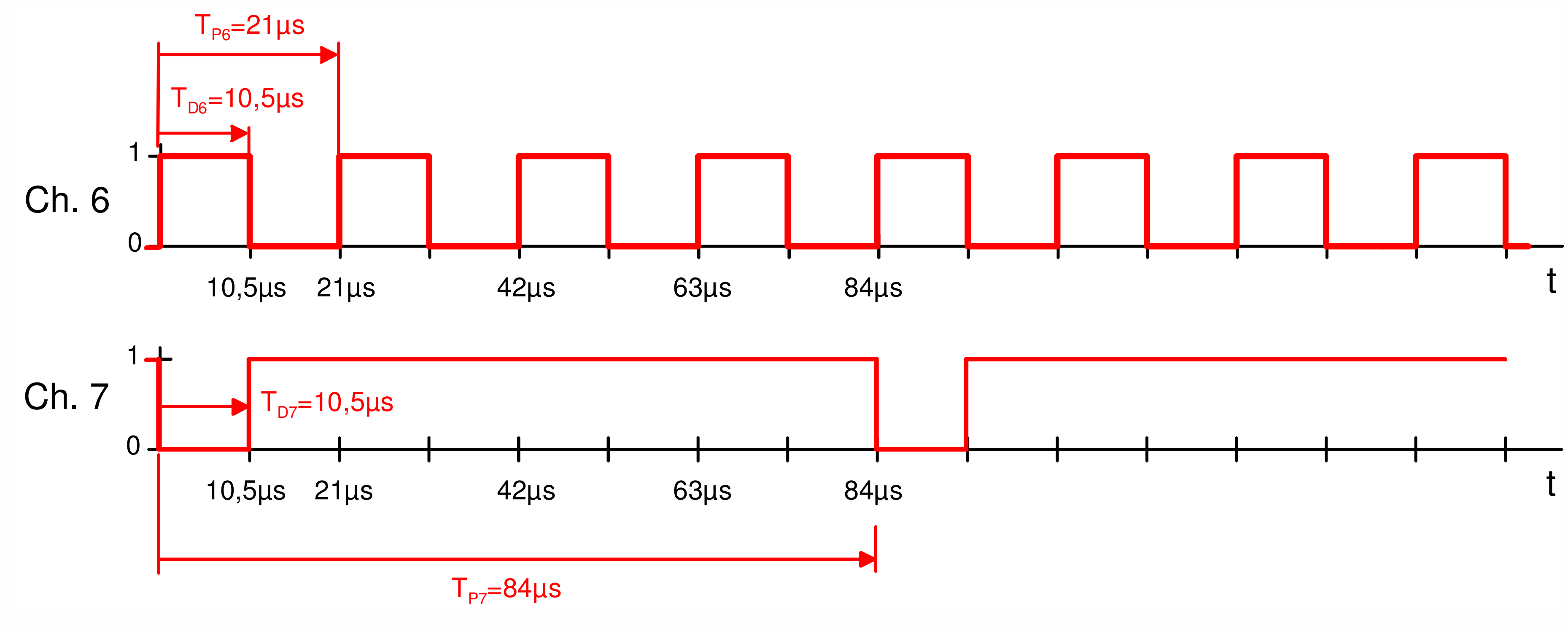
3.5 PWM Outputs

Example: Which PWM signals are generated with the following configuration?
(CodeWarrior project `PWM1.mcp`)

`MOVB #$80, PWMCLK`	Channel 7 uses clock T_{SB} , ch. 6 uses clock T_B
`MOVB #$10, PWMPRCLK`	Divider $x_B = 1$ → Clock period $T_B = 2^1 \cdot 1/24\text{MHz} \approx 83{,}3\text{ns}$
`MOVB #$02, PWMSCLB`	Divider $y_{SB} = 2$ → Clock period $T_{SB} = 2 \cdot 2 \cdot 83{,}3\text{ns} \approx 333\text{ns}$
`MOVB #$40, PWMPOL`	Ch. 7 start with L-phase, ch. 6 starts with H-phase
`MOVB #255, PWMPER6`	Ch. 6: PWM signal period $T_{P6} = 255 \cdot 83{,}3\text{ns} \approx 21\mu\text{s}$
`MOVB #128, PWMDTY6`	Ch. 6: PWM signal H-phase $T_{D6} = 128 \cdot 83{,}3\text{ns} \approx 10{,}5\mu\text{s} = 50\% \cdot T_{P6}$
`MOVB #255, PWMPER7`	Ch. 7: PWM signal period $T_{P1} = 255 \cdot 333\text{ns} \approx 84\mu\text{s}$
`MOVB #32, PWMDTY7`	Ch. 7: PWM signal L-phase $T_{D7} = 32 \cdot 333\text{ns} \approx 10.5\mu\text{s} = 12{,}5\% \cdot T_{P7}$
`BSET PWME,#$C0`	Enable PWM ch. 6 and 7, i.e. turn on the pulse signals

3.5 PWM Outputs

Timing Diagram:



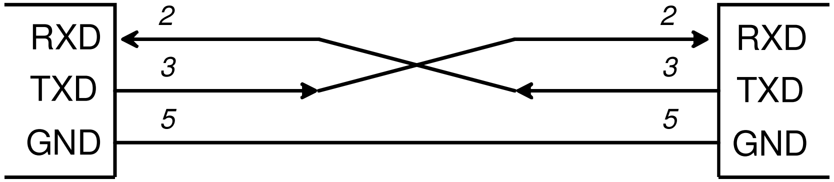
Note:
The PWM can be configured in such a way, that channels 0+1, 2+3, 4+5, 6+7 are combined into four PWM-channels. Thus the duty cycle resolution can be doubled to 16bit.

Rather than starting with the L- (or H-phase), the PWM signal can be generated symmetrically within its period (*Center Aligned*).

3.6 Serial Interface

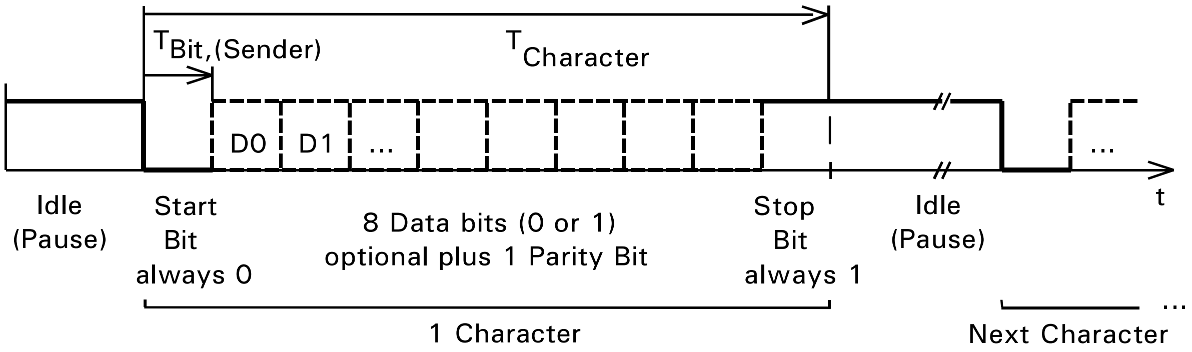
(see Serial Communications Interface SCI [3.6])

- Simple full-duplex communication between two computers:



Subminiatur D-9 Connector Pins

- Data transfer is byte-wise („**Character**"). The communication line is at 1 when idle. Each character begins with a **Start Bit**, which is always 0.
- A character typically consists of **8 Data Bits**, beginning with the LSB.
- **Optionally** followed by a **Parity Bit** (even or uneven parity of the 8 data bits).
- The character transmission ends with a **Stop Bit** (always 1). This level remains constant, till the next character transmission starts (transmission line idle).



Baud rate = Bit rate

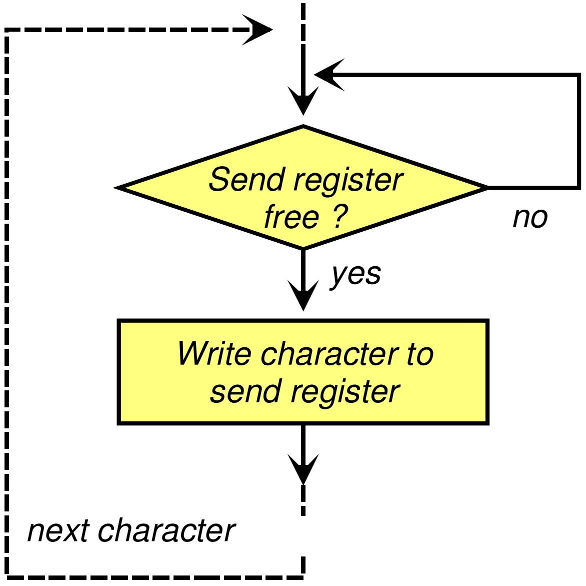
$$f_{bit} = \frac{1}{T_{bit}}$$

Standardized bit rates 9.6, 19.2, 38,4, …, 115.2 kbit/s.

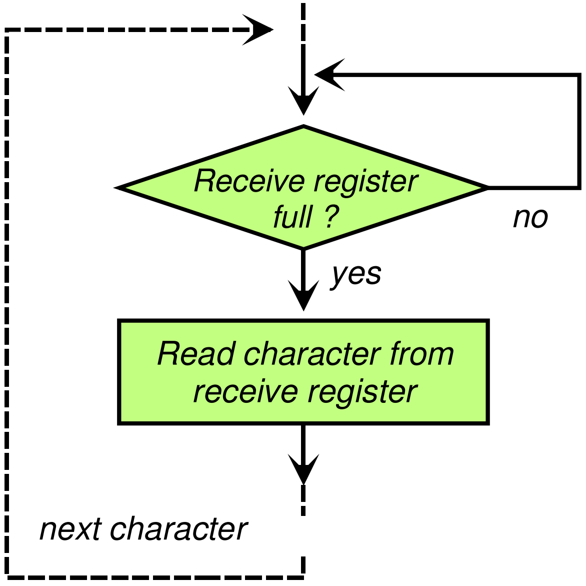
Non-standardized faster bit rates possible.

3.6 Serial Interface

- Sender and receiver must use the same bit rate and the same character format (e.g. 8N1 = 8 data bits, no parity, 1 stop bit). Sender and receiver do not have a common clock signal. Due to slightly different clock frequencies (asynchronous data transmission), only small bit rates are possible.
- The hardware unit, which transmits a single character automatically, is called a **UART Universal Asynchronous Receiver and Transmitter** or **SCI Serial Communication Interface**. If software writes a character into the transmit register of the UART, start, data and stop bits are shifted out to the receiver by the hardware. When a character has been received, software must read it from the UART's receive register. The code structure for data transmission in **Polling Mode** is as follows:



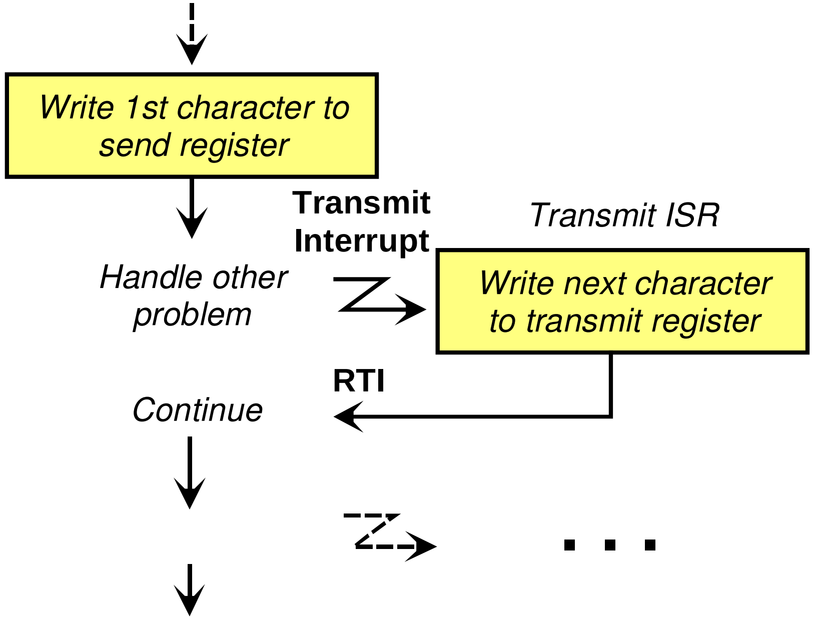
Send character



Receive character

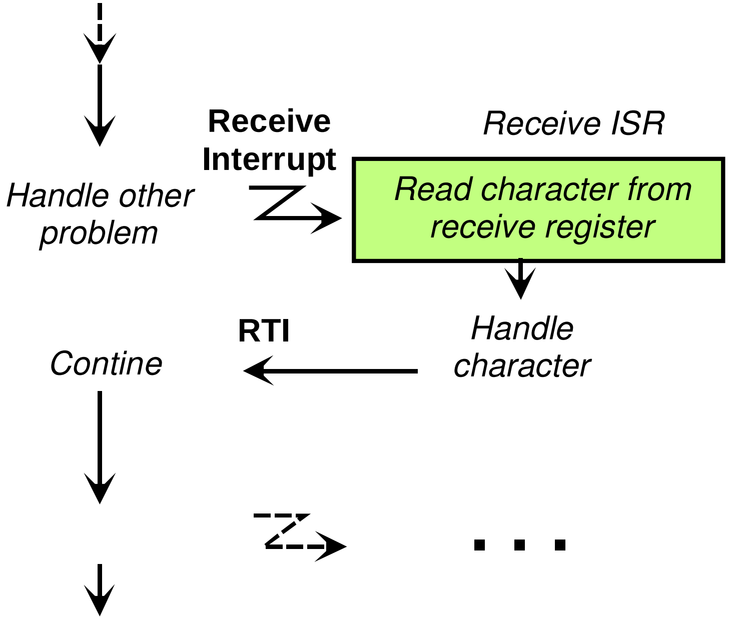
3.6 Serial Interface

- Before writing a character into the transmission register, the software must make sure, that it does not overwrite the previous character. When reading a character from the receive register, the software must make sure, that it does not read a character which it already read before. Rather than polling the status register, an interrupt may be used. The interrupt does signal, when a character has been received (**Receive Interrupt**, used by most programs, because there may be long pauses between characters) and/or that the transmission register is free again, when a character has been sent (**Transmit Interrupt,** may or may not be used).



Write 1st character to send register
Transmit Interrupt
Transmit ISR
Handle other problem
Write next character to transmit register
RTI
Continue
. . .

Send character



Receive character

3.6 Serial Interface

The HCS12 of the Dragon12 board has two UARTs SCI0 and SCI1. SCI0 is used for the debugger communication. SCI1 is free for user programs, e.g. to communicate with a terminal program (Hyperterminal or terminal-component of the HCS12 debugger) running on a PC.

Each serial interface has 3 configuration registers, which must be configured once:

• Baud Rate Register (x=0 for SCI0, x=1 for SCI1)	**SCIxBD** (16bit Register!)	Clock divider $SCIxBD = \frac{f_{BUSCLK}}{16 \cdot f_{bit}}$ (Dragon12 @ f_{BUSCLK}=24MHz)
• Control Register 1	**SCIxCR1** (8bit Register)	Default after reset: 8N1 (8 data bits, no parity, 1 stop bit) If parity is required: Bit 1=1 … Use parity bit Bit 0=1 … Uneven parity (0=odd parity) Bit 7…2 … Do not change
• Control Register 2	**SCIxCR2** (8bit Register)	Sender and receiver control Bit 2 = 1 … Receiver Enable Bit 3 = 1 … Transmitter (Sender) Enable Bit 5 = 1 … Receive Interrupt Enable Bit 7 = 1 … Transmit Interrupt Enable Set other bits to 0 Send and receive interrupt use the same interrupt vector. If both types of interrupt are enabled, the ISR must poll status register SCIxSR1 to find out, why the interrupt was triggered.

Sending and receiving uses the same data register:

- Data Register

SCIxDRL (8bit Register)	If written: If read:	TX Data register (for transmission) RX Data register (for reception)
SCIxDRH (8bit Register)	Higher 8 bits of the data register, used only if the SCI is configured for character length > 8 bit	

Polling the status of the serial interface:

- Status Register 1
- Status Register 2

SCIxSR1 (8bit Register)	Bit 7 = 1 … TX data register free Bit 5 = 1 … RX data register full = new character The other status bits indicate various error conditions, e.g. Bit 3 = 1 … Receive register overwritten by next character, before the previous character was read. Bit 0 = 1 … Parity error
SCIxSR2 (8bit Register)	Optional for operating modes not described here.

Due to implementation details of the hardware, the status register SCIxSR1 should always be read, before reading or writing characters from/to the data register. This will automatically clear the status flags for polling and interrupt mode.

3.6 Serial Interface

Example: Sending and receiving in Polling Mode (CodeWarrior project `SerialPolling.mcp`)

```
SCIxBD: EQU   SCI1BD          ; On Dragon12 use SCI1, in the simulator use SCI0
SCIxCR1:EQU   SCI1CR1
CR:     EQU   $13
LF:     EQU   $10
. . .
.const:SECTION                ; ROM: Constant data
message1: DC.B  "Please enter a character", CR, LF, 0
. . .
.init: SECTION                ; ROM: Code section
main:
. . .                         ; Initialize the serial interface
    MOVW #13, SCIxBD          ; Set baud rate 115200 bit/sec
    MOVB #0,  SCIxCR1         ; Default format 8 data bits, no parity, 1 stop bit
    MOVB #$0C,SCIxCR2         ; Enable receiver and transmitter, no interrupts

    LDX  #message1            ; Send string to SCI
    JSR  puts
    JSR  getch                ; Get character from SCI, returns character in B
    JSR  putch                ; Send character in B to SCI
    . . .
```

Sending and receiving is handled in subroutines **`puts`**, **`getch`** and **`putch`**.

3.6 Serial Interface

```
getch:  ; --- Read a character from serial interface, return in B -----------
    BRCLR SCIxSR1, #$20, getch ; Check 'Receive Data Flag' and wait
    LDAB SCIxDRL               ; Read received character
    RTS                        ; ... and return in B

putch:  ; --- Send a character in B to serial interface --------------------
    BRCLR SCIxSR1, #$80, putch ; Check 'Transmit Register Empty' and wait
    STAB SCIxDRL               ; Send character
    RTS                        ; ... and return

puts:   ; --- Send string to serial interface, X is a pointer to the string -
    BRCLR SCIxSR1, #$80, puts  ; Check 'Transmit Register Empty' and wait
    MOVB 0, X, SCIxDRL         ; Send one character
    TST  1, X+                 ; Check for end of the string …
    BNE  puts                  ; if no, go to the next character
    RTS                        ; Send complete, return
```

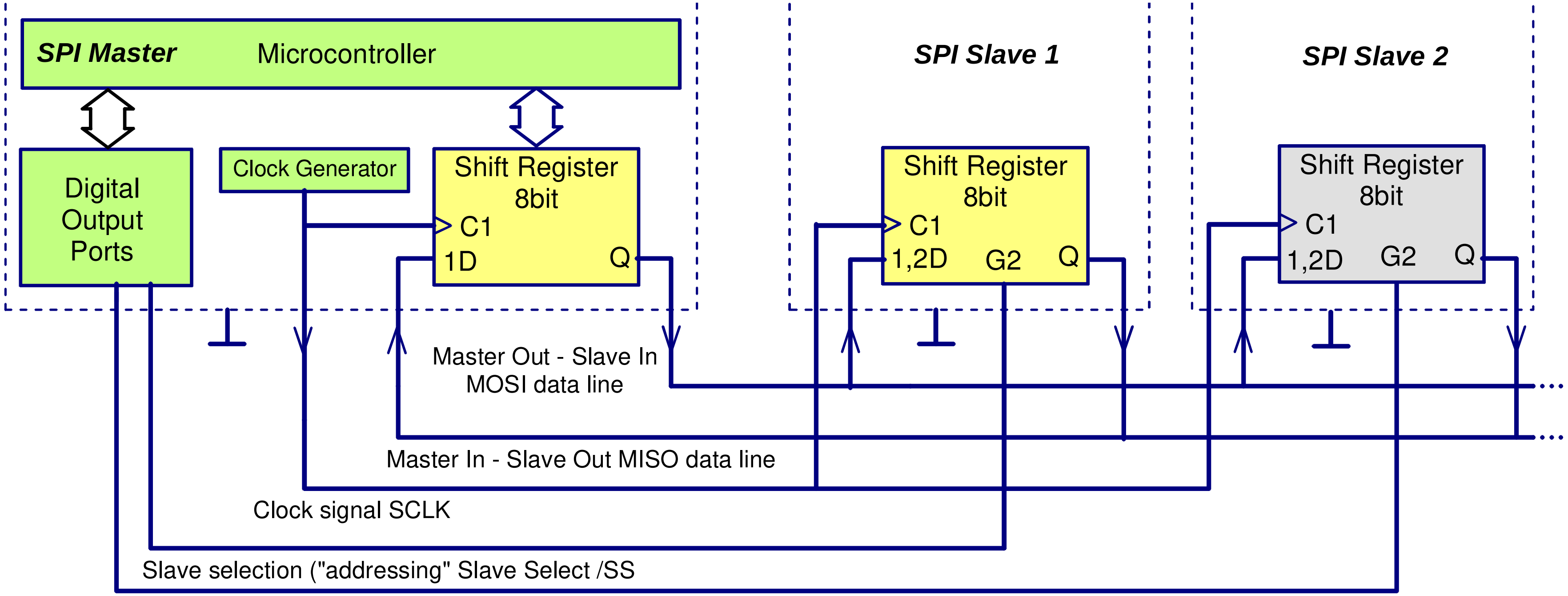
A program version using interrupts rather than polling can be found in CodeWarrior project `serialInterrupt.mcp`.

Note:
User programs must use the second serial interface SCI1 on the Dragon12 board, because the first interface SCI0 is used by the debugger monitoring program. In the simulator, user programs must use SCI0, because the simulator does not support SCI1.

To enable SCI1 on new Dragon12 boards, jumper J23 must be set to position RS232 (already done in our lab, but on new boards factory setting is SCI1 disabled).

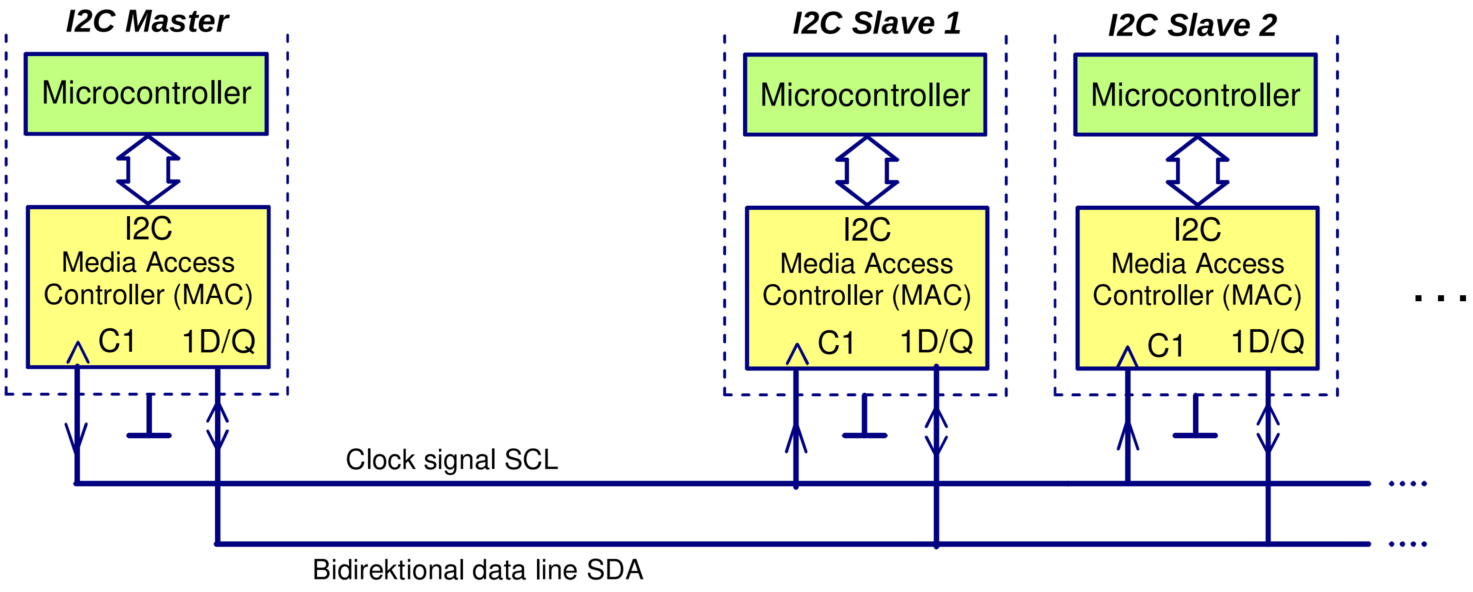
Serial Peripheral Interface SPI

- Serial interface between a microcontroller and peripheral ICs on a printed circuit board PCB
- Industry standard (Freescale and others) to connect serial EEPROMs or external A/D- and D/A-converter ICs
- Basic principle: ‚Ring of shift registers', clocked by **SPI Master** with up to 4 Mbit/s
- Byte wise data transmission clocked by master (1 byte Master → Slave, at the same time 1 byte Slave → Master), duplex communication, but **SPI Slaves** cannot start a transmission, no acknowledge or error signaling from receiver to sender
- Master selects one slave per transmission via a separate "address line" per slave



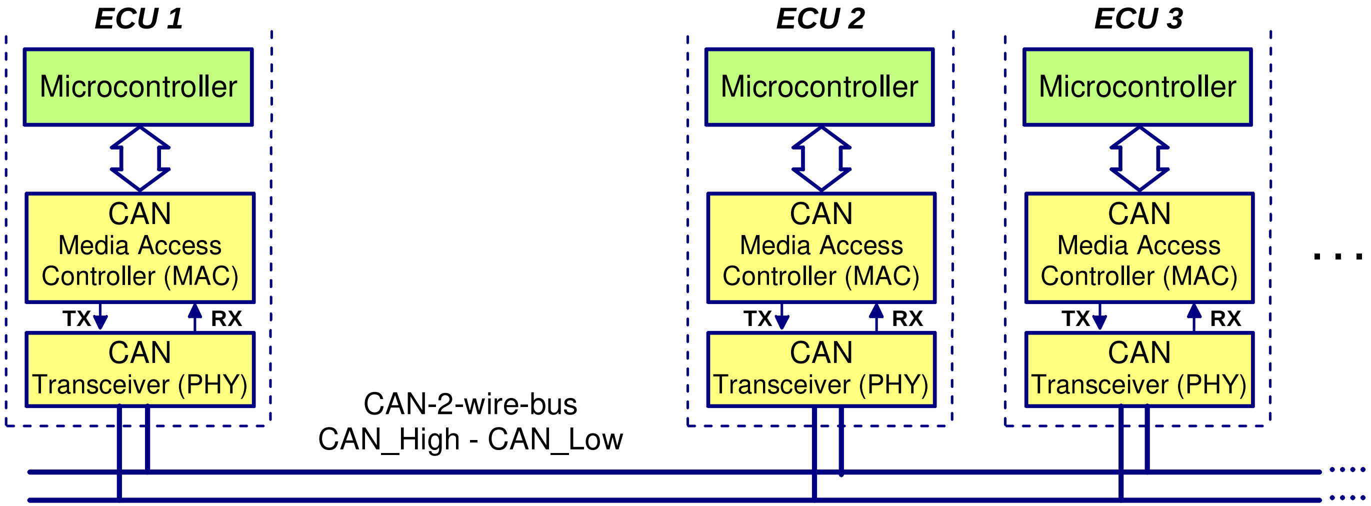
Inter IC Bus I²C

- Serial interface between a microcontroller and peripheral ICs on a printed circuit board PCB
- Industry standard (Philips/NXP and others), originated in consumer electronics, e.g. TV sets
- Bidirectional **Master-Slave** system, clocked by Master with up to 400 kbit/s. Limited multi-master capability.
- Messages with 7bit-address header to select one of the slaves and an arbitrary number of data bytes. If data shall be sent from one of the slaves to the master or to other slaves, the Master sends the slave's address. Then the Master stops (but still outputs a clock signal) and the Slave continues with sending the data bytes.
- Each received byte is confirmed by the receiver with an Acknowledge Bit.



Controller Area Network CAN

- Serial interface of automotive and automation Electronic Control Units ECUs. Developed by Bosch, industry standard ISO 11898, licensed to most microcontroller suppliers.
- Multi-Master bus with **bit rates up to 1 Mbit/s**, in cars typically f_{bit}=500 kbit/s (High-Speed-CAN) and f_{bit}=100 kbit/s (Low-Speed-CAN) with different physical layers, uses two wires with differential voltage signals. Due to arbitration (see below) the maximum bus length L limited by bit rate f_{bit}: $L \leq 40\text{m} \cdot 1\text{Mbit/s} / f_{bit}$

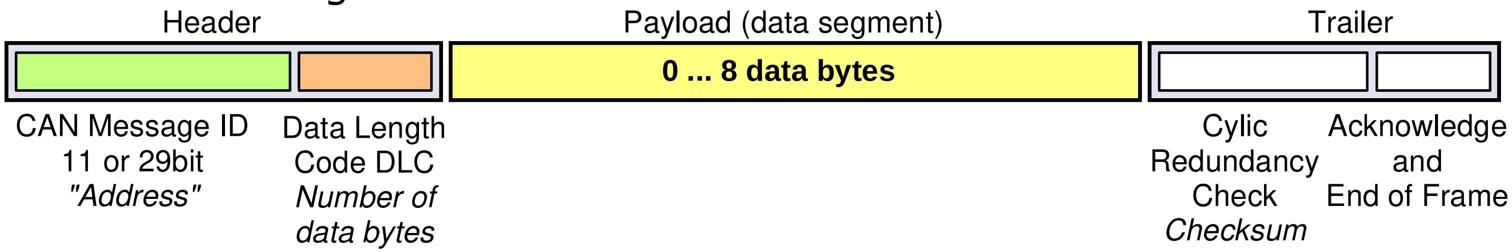


- Messages with 11bit (or 29bit) Addresses (**CAN Identifier**), up to 8 data byte and CRC check sum. Receiver acknowledges reception, automatic retransmission on errors.
- **Content based** rather than station base **addressing**: Receiving ECUs are selecting messages based on their contents. This **acceptance filtering** via the CAN Identifier is done by hardware in CAN communication controllers. (Note: Ethernet MAC addressing is station based).

3.7 Miscellaneous Interfaces SPI, I²C, CAN

- The CAN Identifier is additionally used for message **arbitration**, i.e. if two ECUs start sending at the same time, a bus collision will be detected (because the electrical signal on the bus will be corrupted). The CAN controllers will detect the collision and the message with the smaller valued **CAN identifier** (=higher **priority**) will continue, while the other message will be stopped and resent later, when the bus is idle again (CSMA/CR Carrier Detect Multiple Access/Collision Resolution)

Format of a CAN message



(Various control bits in the header and trailer not shown here, message format automatically generated by hardware).

Simple HCS12 CAN driver (CAN bit rate 250 kbit/s, 11bit CAN identifier):

```
typedef struct                               // Data structure for CAN messages
{    unsigned long CanId;                    // CAN identifier
     unsigned char DataLength;               // Data length (max. 8)
     unsigned char Data[8];                  // Data bytes
} CANMESSAGE;

CANMESSAGE sendMessage, receiveMessage;      // Variables to send and receive messages

void CanInit(void);                          // Functions called by user programs
void CanSendMessage(CANMESSAGE *pMsg);
void CanReadMessage(CANMESSAGE *pMsg);
```

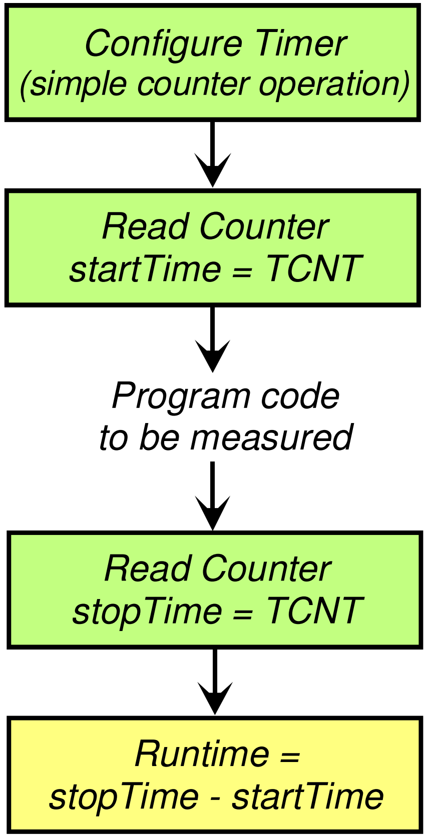
3.7 Miscellaneous Interfaces SPI, I^{2}C, CAN

```
void CanInit(void)                           // CAN controller initialization
{    CAN0CTL0 = CAN0CTL0 | 0x01;             // Request initialization
     while ((CAN0CTL1 & 0x01) ==0) {};       // Wait for initialization start acknowledge
     CAN0CTL1  = 0x80;                       // CAN enable, use OSCCLK (Quartz-Takt)
     CAN0BTR0  = 0xC1;                       // 250 kbit/s @ fOSCCLK = 8MHz (0xC0 bei fOSCCLK=4MHz)
     CAN0BTR1  = 0x58;
     CAN0IDAC  = 0x20;                       // Receive CAN messages with any CAN identifier
     CAN0IDAR0 = 0x00;
     CAN0IDMR0 = 0xFF;
     CAN0CTL0 = CAN0CTL0 & 0xFE;             // End initialization
     while ((CAN0CTL1 & 0x01) == 1) {};      // Wait for end of initalization acknowledge
     CAN0RFLG_RXF = 1;                       // Clear receiver flags
     CAN0RIER = CAN0RIER | 0x01;             // Receive interrupt enable
}
void CanSendMessage(CANMESSAGE *pMsg)        // Send a CAN message (typ. pMsg = &sendMessage)
{    volatile unsigned char txBuf;
     while ((CAN0TFLG & 0x07) == 0) {};      // Wait for a free transmit buffer
     txBuf = CAN0TFLG & 0x07;                // Select the transmit buffer
     CAN0TBSEL = txBuf;
     txBuf = CAN0TBSEL & 0x07;
     CAN0TXIDR0 = (unsigned char) (pMsg->CanId >> 3); // Copy message to transmit buffer
     CAN0TXIDR1 = (unsigned char) (pMsg->CanId << 5); //   -- here: CAN identifier
     CAN0TXIDR2 = 0x00;
     CAN0TXIDR3 = 0x00;
     memcpy (&CAN0TXDSR0, pMsg->Data, pMsg->DataLength);// -- here: CAN data
     CAN0TXDLR  = pMsg->DataLength;          // Set data length (number of bytes in message, max.8)
     CAN0TXTBPR = CAN0TXIDR0;
     CAN0TFLG = txBuf;                       // Send the message
}
```

3.7 Miscellaneous Interfaces SPI, I^{2}C, CAN

```
void CanReadMessage(CANMESSAGE *pMsg)             // Receive a CAN message (polling or used in ISR)
{   unsigned long temp;                              (typ. pMsg = &receiveMessage)

    if ((CAN0RFLG & 0x01)==0)                     // Return at once, if the receive buffer is empty
    {   pMsg->DataLength = 0;
        return;
    }

    pMsg->CanId = CAN0RXIDR0;                     // Copy CAN identifier to CAN data structure
    pMsg->CanId = (pMsg->CanId << 3) + (CAN0RXIDR1 >> 5);

    (void) memcpy(pMsg->Data, &CAN0RXDSR0, CAN0RXDLR);// Copy CAN data bytes to CAN data structure
    pMsg->DataLength = CAN0RXDLR;                 // Copy CAN data length

    CAN0RFLG = CAN0RFLG | 0x01;                   // Clear receive flag
}

void interrupt 38 CanReceiveISR(void)             // CAN message receive ISR
{   CanReadMessage(&receiveMessage);              // Copy CAN message to global variable

    // Do whatever is needed with received data or notify user program by setting an event variable
    //  ...

    CAN0RFLG = CAN0RFLG | 0x01;                   // Clear receive flag
}
```

Assembler Version of Program for Run-Time Measurement (CodeWarrior project timer1.mcp)



```
.data: SECTION
startTime DS.W 1        ; TCNT at start of measurement
stopTime  DS.W 1        ; TCNT at end of measurement
runTime   DS.W 1        ; runTime= stopTime - startTime

.init: SECTION
main:
Entry:  . . .

; --- Initialize timer ---------------------------------
    BSET TSCR1, #$80    ; Enable timer module
    MOVB #$07, TSCR2    ; Timer clock period 2^7/24MHz

    MOVW TCNT,startTime ; Save start time

; --- Program code to be measured ----------
    . . .

; --- Compute run time ---------------------------------
    MOVW TCNT, stopTime ; Save stop time

    LDD  stopTime       ; Compute run time
    SUBD startTime
    STD  runTime        ; Einheit: Timer clock period
```

Note: Resolution is 1 timer clock period, i.e. 5.3µs   measurement range: 1 counter period, i.e. max. 350ms

A smaller clock period will increase measurement resolution, but will reduce the measurement range.

Assembler Version of Program to Measure the Period of a Pulse Signal

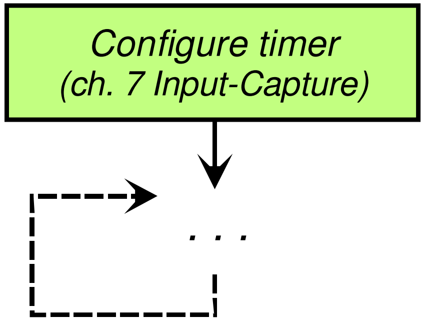
The pulse signal is connected to port T.7 (used as input) and shall have a frequency range of 10Hz … 10kHz.

(CodeWarrior project timer3.mcp)

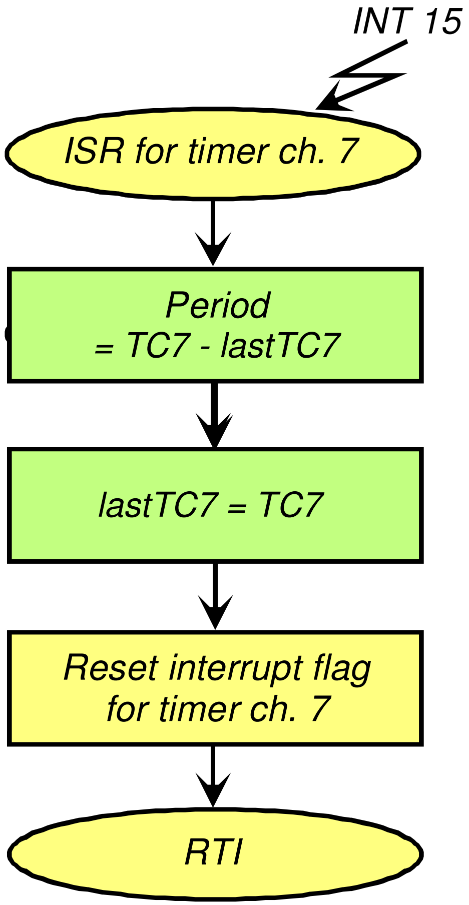
```
.data: SECTION
signalPeriod: DS.W 1    ; Signal period in TCNT clock
                        ; periods
lastTC7:      DS.W 1    ; TC7 at last input event

.vect: SECTION          ; ROM: Interrupt vector entries
       ORG $FFE0
       DC.W timer7Isr   ; Timer channel 7 interrupt

.init: SECTION
main: . . .
```

Main program:

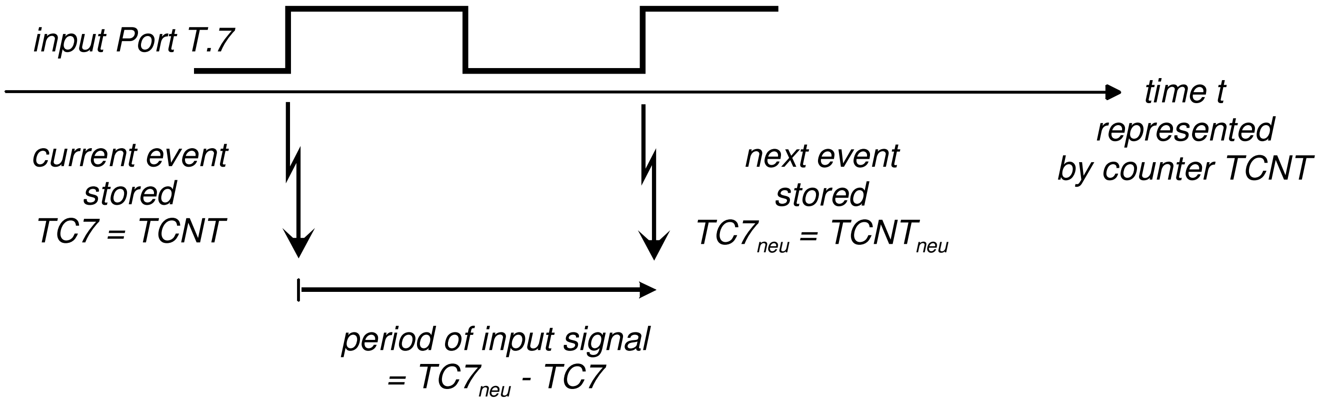
```
; --- Initialize timer --------------------------------
    BSET TSCR1, #$80    ; Enable timer module
    MOVB #$07, TSCR2    ; Timer clock period 2^7/ 24MHz
    BCLR TIOS, #$80     ; Timer ch. 7: input capture
    MOVB #$40, TCTL3    ; Trigger ch. 7 on rising edge
    BSET TIE,  #$80     ; Enable interrupt for ch. 7


loop:   BRA  loop       ; Infinite loop
```



Appendix A: Assembler Version of several Sample Programs



```
; --- Interrupt service routine for timer channel 7 ---
timer7Isr:
    LDD  TC7             ; Get current TC7
    PSHD

    SUBD lastTC7         ; Compute signalPeriod
    STD  signalPeriod    ;   = current TC7 – last TC7

    PULD                 ; Save current TC7
    STD  lastTC7         ;   for next measurement

    BSET TFLG1, #$80     ; Reset interrupt flag of ch. 7

    RTI
```



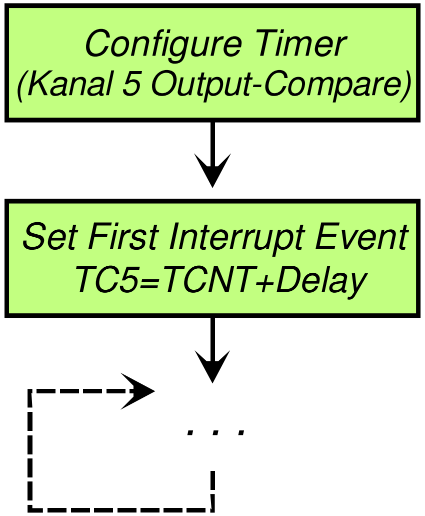
"time" as integer multiple of counter clock period TCNT

Assembler Version to Drive the Dragon12 Beeper in Output-Compare Mode

A beeper (buzzer) on port T.5 (used as output) shall be driven by a 500Hz pulse signal with 1:1 duty cycle:

(CodeWarrior project timer2.mcp)

Main program:



Configure Timer (Kanal 5 Output-Compare)

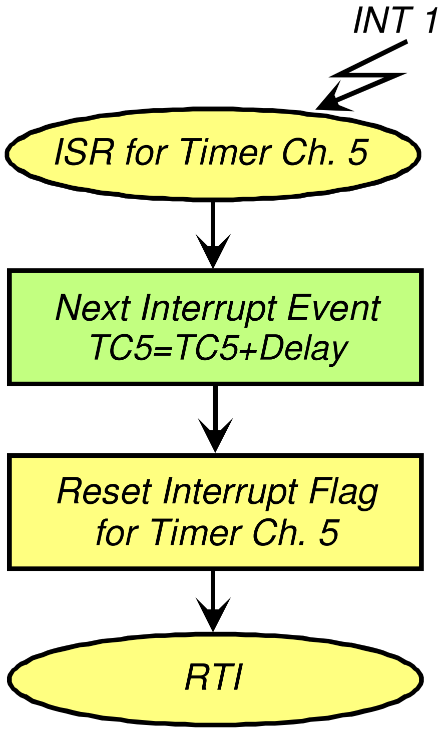
Set First Interrupt Event TC5=TCNT+Delay

. . .

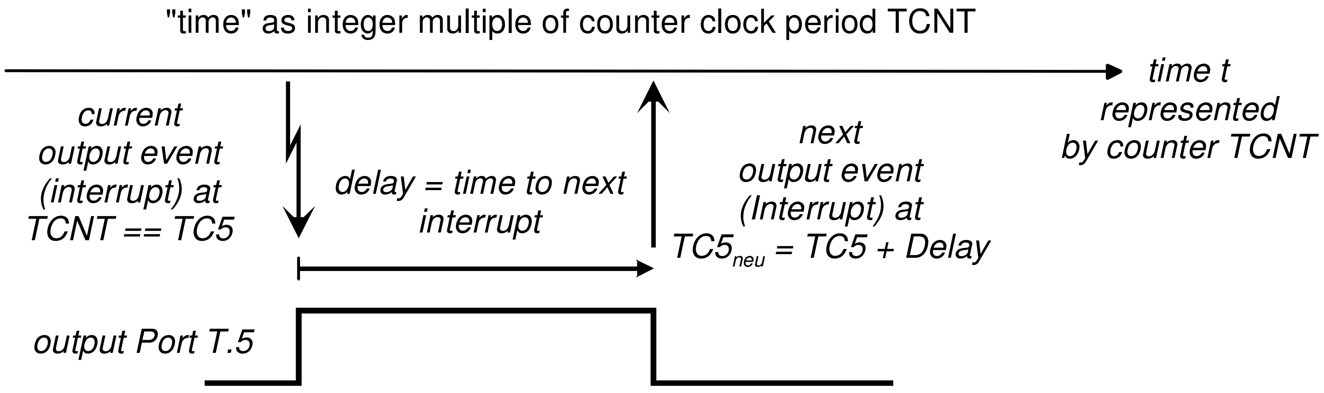
```
DELAY: EQU (24000/128)  ; Delay 1ms * 24Mhz / 2^7
. . .
.vect: SECTION          ; ROM: Interrupt vector entries
       ORG $FFE4
       DC.W timer5Isr   ; timer channel 5 interrupt

.init: SECTION
main:. . .
; --- Initialize timer --------------------------------
    BSET TSCR1, #$80    ; Enable timer module
    MOVB #$07, TSCR2    ; Timer clock period 2^7/ 24MHz
    BSET TIOS, #$20     ; Timer ch.5 as output compare
    MOVB #$04, TCTL1    ; On timer event toggle output

    LDD  TCNT           ; Set timer for first interrupt
    ADDD #DELAY
    STD  TC5

    BSET TIE,  #$20     ; Enable interrupt for ch. 5

lp: BRA  lp             ; Infinite loop
```

Appendix A: Assembler Version of several Sample Programs

ISR:



```
; --- Interrupt service routine for timer channel 5
; Output port T.5 is toggled automatically, whenever
; this ISR is called

timer5Isr:

    LDD  TC5              ; Set timer for next interrupt
    ADDD #DELAY           ;(Don’t use TCNT here to reduce jitter)
    STD  TC5

    BSET TFLG1, #$20      ; Reset interrupt flag of ch.5

    RTI
```



By setting the bits in TCTL1 to 0, the output compare mode will trigger an ISR only at a predefined time, but does not change the port pins output signal.

Assembler Version of Program to measure analog signal

Example: (CodeWarrior project `ADCInterrupt.mcp`)

- Measure the analog signal on channel 7 (connected to a poti on Dragon12 boards)
- Output the arithmetic mean of 2 measurement values (in binary) to the LEDs on port B

```
. . .
.data:  SECTION                   ; RAM: Variable data section
value:  DS.W  1                   ; Measurement value 10bit, right justified

.vect:  SECTION                   ; ROM: Interrupt vector entries
        ORG $FFD2
        DC.W adcIsr               ; Interrupt vector for interrupt 22 (ATD0)

.init: SECTION                    ; ROM: Code section
main:
Entry:  . . .
    ; --- Initialize ATD0 ----------------------------------------
    MOVB #%11000010, ATD0CTL2     ; Enable ATD0, enable interrupt
    MOVB #%00010000, ATD0CTL3     ; Sequence of 2 measurements
    MOVB #%00000101, ATD0CTL4     ; 10bit, 2MHz ATD0 clock
    MOVB #%10000111, ATD0CTL5     ; Start first measurement on single channel 7

lp: LDD  value                    ; Show upper 8bits of meas. value on LEDs 7...0
    LSRD
    LSRD
    STAB PORTB
    BRA  lp
```

Appendix A: Assembler Version of several Sample Programs

```
    ; --- Interrupt Service Routine for interrupt 22 -----------------------
adcIsr:
    LDD  ATD0DR0                 ; Read the result registers,
    ADDD ATD0DR1                 ; ... compute average of 2 measurements
    LSRD
    STD  value                   ; ... and store result

    BSET ATD0CTL2, #$01          ; Reset interrupt flag (bit 0 in ATD0CTL2)

    MOVB #%10000111, ATD0CTL5    ; Start next measurement on single channel 7
    RTI                          ; Return from interrupt service routine
```

Note: Result registers ATD0DR0, … must be **read** (to automatically reset the CCF flag), **before** starting the **next conversion.**

Instead of starting the A/D conversion via software (by writing to ATD0CTL5), the HCS12 allows other modes to trigger a conversion:

- Automatic Triggering: In this mode (scan mode or free running mode, activated via bit 5=1 in register ATD0CTL5) software only triggers the first conversion. All following conversions will start automatically after end of the previous conversion. Result registers can be read any time and always provide the latest result. However, in this mode sampling is asynchronous, i.e. the user program has no control of the signal sampling period.
- Hardware Triggering: Rather than by software, a conversion will be started by an external hardware signal (a rising or falling edge of an pulse signal at channel 7 of the ADC).

Appendix B: Clock Generator and Clock Divider Settings

If the Dragon12 board shall be used without the debugger monitor program, the user program must initialize the CPU's clock generator. After Reset, the CPU's clock is directly driven by the quartz crystal @ 4MHz. With the debugger, the monitor program does the initialization.

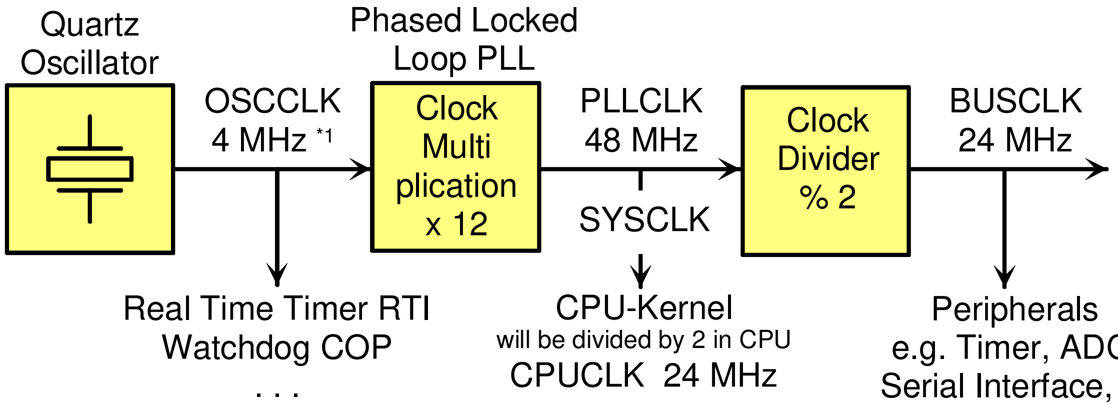
To set the clock generator to a higher frequency, the following code sequence can be used (see Clock & Reset Generator Module CRG [3.4]). This code assumes quartz with f_{OSCCLK}=4MHz [*1] and sets the PLL for a bus clock of f_{BUSCLK}=24MHz (max. for Dragon12 boards):

```
        BCLR CLKSEL, #$80           ; Disconnect PLL from CPU (only in case)
        BSET PLLCTL, #$40           ; Turn on PLL
        MOVB #$05,SYNR              ; Set PLL multiplier
        MOVB #$00,REFDV             ; Set PLL divider (*1 Dragon12 Plus 8MHz: #$01)
        NOP
        NOP
pllWait:BRCLR CRGFLG, #$08, pllWait ; Wait till PLL has locked
        BSET  CLKSEL, #$80          ; Connect PLL to CPU
         . . .
```

$$f_{PLLCLK} = 2\, f_{OSCCLK} \frac{SYNR+1}{REFDV+1}$$

max. 48MHz

$$f_{BUSCLK} = \frac{f_{PLLCLK}}{2}$$


Quartz Oscillator

OSCCLK 4 MHz [*1]

Phased Locked Loop PLL

Clock Multi plication x 12

PLLCLK 48 MHz

SYSCLK

Clock Divider % 2

BUSCLK 24 MHz

Real Time Timer RTI
Watchdog COP
. . .

CPU-Kernel
will be divided by 2 in CPU
CPUCLK 24 MHz

Peripherals
e.g. Timer, ADC,
Serial Interface, ...

[*1] Note:
OSCCLK
Dragon12:
4MHz
Dragon12Plus:
8MHz

Appendix B: Clock Generator and Clock Divider Settings

RTI Interrupt

Interrupt period $T_{RTI} = 1/f_{RTI} = 2^{9+X}\cdot(Y+1) / f_{OSCCLK.}$ All values in ms @ $f_{OSCCLK} = 4\text{MHz}$

	Y= 0	Y= 1	Y= 2	Y= 3	Y= 4	Y= 5	Y= 6	Y= 7	Y= 8	Y= 9	Y=10	Y=11	Y=12	Y=13	Y=14	Y=15
X= 1:	0.256	0.512	0.768	1.024	1.280	1.536	1.792	2.048	2.304	2.560	2.816	3.072	3.328	3.584	3.840	4.096
X= 2:	0.512	1.024	1.536	2.048	2.560	3.072	3.584	4.096	4.608	5.120	5.632	6.144	6.656	7.168	7.680	8.192
X= 3:	1.024	2.048	3.072	4.096	5.120	6.144	7.168	8.192	9.216	10.240	11.264	12.288	13.312	14.336	15.360	16.384
X= 4:	2.048	4.096	6.144	8.192	10.240	12.288	14.336	16.384	18.432	20.480	22.528	24.576	26.624	28.672	30.720	32.768
X= 5:	4.096	8.192	12.288	16.384	20.480	24.576	28.672	32.768	36.864	40.960	45.056	49.152	53.248	57.344	61.440	65.536
X= 6:	8.192	16.384	24.576	32.768	40.960	49.152	57.344	65.536	73.728	81.920	90.112	98.304	106.496	114.688	122.880	131.072
X= 7:	16.384	32.768	49.152	65.536	81.920	98.304	114.688	131.072	147.456	163.840	180.224	196.608	212.992	229.376	245.760	262.144

ECT Timer @ $f_{BUSCLK} = 24\text{MHz}$

Clock period $T_{TCNT} = 1/f_{TCNT} = 2^X / f_{BUSCLK}$. values in µs

X= 0	X= 1	X= 2	X= 3	X= 4	X= 5	X= 6	X= 7
0.042	0.083	0.167	0.333	0.667	1.333	2.667	5.333

Counter period $T_P = 2^{16} \cdot T_{TCNT}$. values in ms

X= 0	X= 1	X= 2	X= 3	X= 4	X= 5	X= 6	X= 7
2.731	5.461	10.923	21.845	43.691	87.381	174.763	349.525

PWM @ $f_{BUSCLK} = 24\text{MHz}$

Clock period of fast clock $T_{A/B} = 2^X / f_{BUSCLK}$, values in µs

X= 0	X= 1	X= 2	X= 3	X= 4	X= 5	X= 6	X= 7
0.042	0.083	0.167	0.333	0.667	1.333	2.667	5.333

Clock period of slow clock $T_{SA/B} = 2 \cdot y \cdot T_{A/B}$. values in µs

	X= 0	X= 1	X= 2	X= 3	X= 4	X= 5	X= 6	X= 7
Y= 1:	0.083	0.167	0.333	0.667	1.333	2.667	5.333	10.667
Y= 2:	0.167	0.333	0.667	1.333	2.667	5.333	10.667	21.333
. . .	. . .							
Y=255:	21.250	42.500	85.000	170.000	340.000	680.000	1360.000	2720.000
Y=0(=256):	21.333	42.667	85.333	170.667	341.333	682.667	1365.333	2730.667

Serial Interface @ $f_{BUSCLK} = 24\text{MHz}$

Clock divider register SCIxBD = $f_{BUSCLK} / (16 \cdot f_{Bit})$

f_{Bit}= 300bit/s	600bit/s	1,2kbit/s	2,4kbit/s	4,8kbit/s	9,6kbit/s	19,2kbit/s	38,4kbit/s	57,6kbit/s	115,2kbit/s
5000	2500	1250	625	313	156	78	39	26	13

Chapter 4

Modular Programming in C and Assembler

4.1 Introduction

Most professional embedded system programs are written in C, sometimes C++, because coding in C is faster and programs are more maintainable than assembler code. Nevertheless, a small part of these programs often will be programmed in other languages, especially in assembler, because

- **Algorithms in assembler** can use special features of the CPU's instruction set, which a C-compiler may not use. Thus assembler code may **run faster** and/or require **less memory**. Less advanced C-compilers, especially those, which support different CPU families, typically use only those CPU instructions, which are available on most microprocessor types. Special commands like BIT SET/CLEAR, which may be much faster than AND/OR operations, or instructions, which add and multiply in a single operation like HCS12's EMACS instruction, may not be used, because they are not available in all types of CPUs.
- **Accessing special registers or functions of the microcontroller**, for which no equivalent C-operation is available, i.e. there are no C-statements to access status register CCR directly.

There are two basic solutions for the interaction of C and assembler code:

- **Inline-Assembler**: Embed assembler instructions into C code.
- **Assembler-Module**: Assembler subroutines, which are called from C code.

4.2 Inline-Assembler (see [3.13 Chapter High Level Inline Assembler])

Single assembler instructions can be integrated in C-code as **`asm befehl`**;
For (small) blocks of assembler instructions within a C function use

```
asm { ...
    }
```

Beause the C-compiler has to use the same CPU registers as the assembler, some rules ensure a smoothless interaction between C and assembler code:

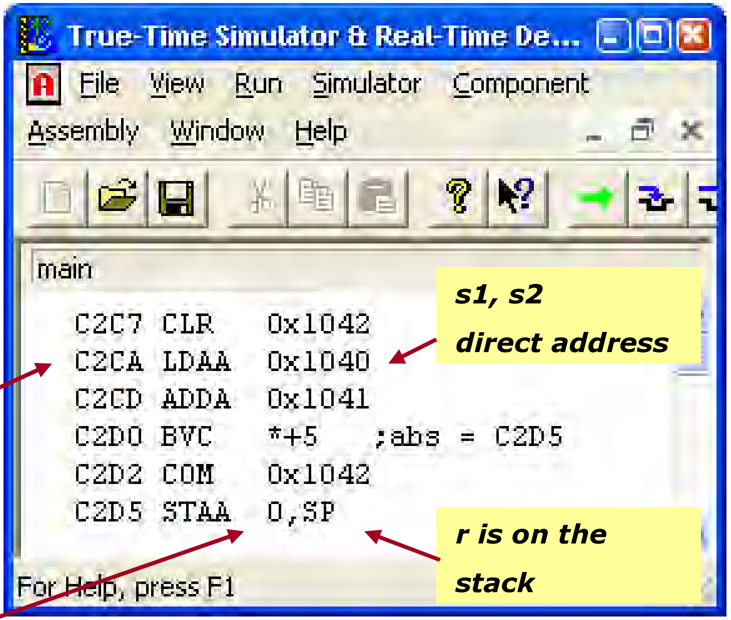
- **Variables and constants** shall be declared **in C**. Assembler code can access these variables by using the variable name. Global C-variables can be used in all assembler instructions which work with direct addressing. Local C-variables require assembler instructions which allow register-indirect addressing via **`SP`**. The C-compiler will handle the translation between the C variable name and the CPU's address mode.
- At the end of each block of assembler instructions, the **Stack Pointer** must have the same value as at the start of the block.
- The address of a C-variable is available as **`@variableName`** in the assembler block.
- Elements of C-structures and unions can be used, e.g. **`LDD myStruct.myElement`**
- Elements of C-arrays can be used, e.g.: **`int a[20]; asm LDD a:24`** accesses element **`a[12]`**. Note: The C-array index must be converted into a byte offset in the assembler instruction, i.e. **`variableName : C-Index · sizeof(variableType)`**

Some compilers allow to freely modify all CPU registers in assembler blocks (e.g. HCS12 CodeWarrior). Other compilers (e.g. GNU C) require that CPU registers be saved on the stack and restored at the end of the assembler block. Check the compiler manual!

Example Program **InlineAsm.mcp**

- Add two 8bit variables and indicate overflow

```
char s1, s2;                  // Global variables
char flag;

void main(void)
{   char r;                   // Local variable
     . . .
    for(;;)
    {   . . .
        s1 = . . .            // Input summands from terminal
        s2 = . . .
        . . .
        flag = 0;

        asm
        {       LDAA s1       // Compute r = s1 + s2, if overflow occurs, set flag=1
                ADDA s2       // Compute sum
                BVC  noov     // Check for overflow
                COM  flag     // ... and set flag if overflow occured
          noov: STAA r        // Store result
        }

        sprintf(temp, "\nc =%d + %d = %d   %s\n", s1, s2, r, flag ? "Overflow" : "No overflow");
        PutString(temp);      // Output result to terminal

        asm SWI;              // Exit program (stop simulator or return to monitor program)
    }
}
```


True-Time Simulator & Real-Time De...
File View Run Simulator Component
Assembly Window Help
main
For Help, press F1

```
C2C7 CLR   0x1042
C2CA LDAA  0x1040
C2CD ADDA  0x1041
C2D0 BVC   *+5    ;abs = C2D5
C2D2 COM   0x1042
C2D5 STAA  0,SP
```

s1, s2 direct address

r is on the stack

4.3 Assembler-Subroutines for C-Programs

(see [3.13 Chapter HC12 Backend – Call Protocol and Calling Conventions and Chapter Stack Frames])

To pass parameters to a subroutine and return results, the caller and the callee must use a common strategy and data types for parameter passing:

```
    returnDatatype function(C_datatype1 param1, . . ., C_datatypeN paramN)
```

Several methods exist:

- **Use global variables**: Advantage: No size limit (other than RAM size). Disadvantage: Recursive subroutine calls (Reentrancy) impossible.
- **Use registers**: Advantage: Fast. Disadvantage: Limited size/number of registers. This method is mainly used in stand-alone assembler programs.
- **Use the stack**: Advantage: Flexible. Disadvantage: Slow, error-prone addressing of parameters.

Most C-compilers use a mix of parameter passing via registers and via the stack. Unfortunately, these **Calling Conventions** are not standardized, so each programming language and each compiler comes up with its own solution. The **CodeWarrior HCS12-C-Compiler** uses the following methods:

- The **result** of a function is **returned in** a **register** (D, see table on the next page).
- If a function has a fixed number `N` of parameters: **Parameters `1` to `N-1` passed on the stack „from left to right**" (PASCAL calling convention). However, the **last parameter `paramN`** is passed **in** a **register** (D, see table).

4.3 C with Assembler Subroutines

Data type of last parameters *Data type of return value*	*Size in byte*	*CPU-register*
char	1	**B**
int (Near) Pointer **`*any_C_datatype`**	2	**D**
long	4	**X** (upper 2Byte), **D** (lower 2Byte)

Big data structures (arrays, structures, unions) should be passed *by reference*, rather than *by value*, i.e. only a pointer to the data structure is passed (see [3.13])

- The **calling program** has to **remove** the **parameters from stack** again, after the subroutine returned. This is done by modifying **SP** by the number of bytes, which have been passed as parameters on the stack. I.e.

param1 → Stack ("Push")	Parameter passing via stack
. . .	"left to right" (decrements SP)
paramN-1→ Stack	
paramN → Register	Last parameter in register
Call the subroutine	
Store return value	
SP+number of parameter bytes → **SP**	"Clean the stack" (increment SP)

- Note: If a function has a variable number of parameters, e.g. **printf()**, **sprintf()**, …, **all** parameters will be passed on the stack „from right to left", i.e. parameter **paramN** is pushed first, parameter **param1** is pushed last to the stack before calling the subroutine (C calling convention).

4.3 C with Assembler Subroutines

The **interface must be defined** exactly for compiler, assembler and linker to work:

Interface in calling C-program:

- Function prototype

```
C_datatype function(C_datatype1 param1, . . ., C_datatypeN paramN);
```

(Note: The C keyword “extern” is optional, but usage is not recommended.)

- Reference to a global assembler variable (possible, but considered bad style)

```
extern C_datatype assemblerVariable;
```

Interface in called assembler-program:

- **Export** the assembler subroutine to other program modules

```
XDEF nameAsmSubroutine                  // at begin of file
nameAsmSubroutine:  ...                 // label at begin of subroutine code
```

- Export a global assembler variable to other program modules

```
XDEF  assemblerVariable                 // at begin of file
assemblerVariable DS.…  …               // normal declaration of global assembler variable
```

- **Import** of global C-variable

```
XREF  C_variable                        // at begin of file
```

4.3 C with Assembler Subroutines

Example Program **CwithASM.mcp**

- Add two 8bit variables and indicate overflow

Main program in C:

```
char asmCompute(char s1, char s2); // Prototype

char s1, s2, r, flag=0;       // Global variables

void main(void)
{   . . .
    r = asmCompute(s1, s2);   // Call to assembler function
    . . .
}
```

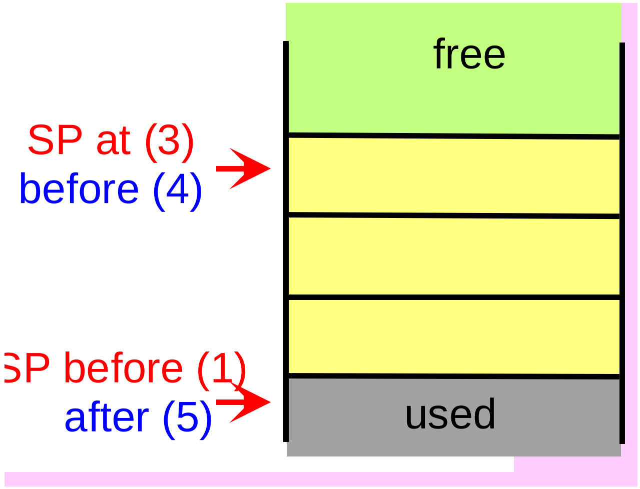
Compiled into

```
LDAB  s1               ; s1 → Stack
PSHB                   ;(1)
LDAB  s2               ; s2 → B
JSR   asmCompute       ;(2)
LEAS  1,+SP     Clean  ;(5) stack
STAB  r              ; Result in R
```

Subroutine in assembler:

```
       XDEF   asmCompute     ; Export of ASM function

       XREF   flag           ; Import of global C variable

.init: SECTION          ; Assembler program code in ROM
asmCompute:    LDAA  2, SP    ; Get s1 from stack      (3)
               ABA            ; s1 + s2 (in reg. B) --> reg. A
               BVC   noov     ; Check for overflow
               COM   flag     ; ... and set global C variable
                              ;     (bad programming style!)
noov:          TFR   A, B     ; Move result to B
               RTS            ; Return to caller       (4)
```

State of stack:

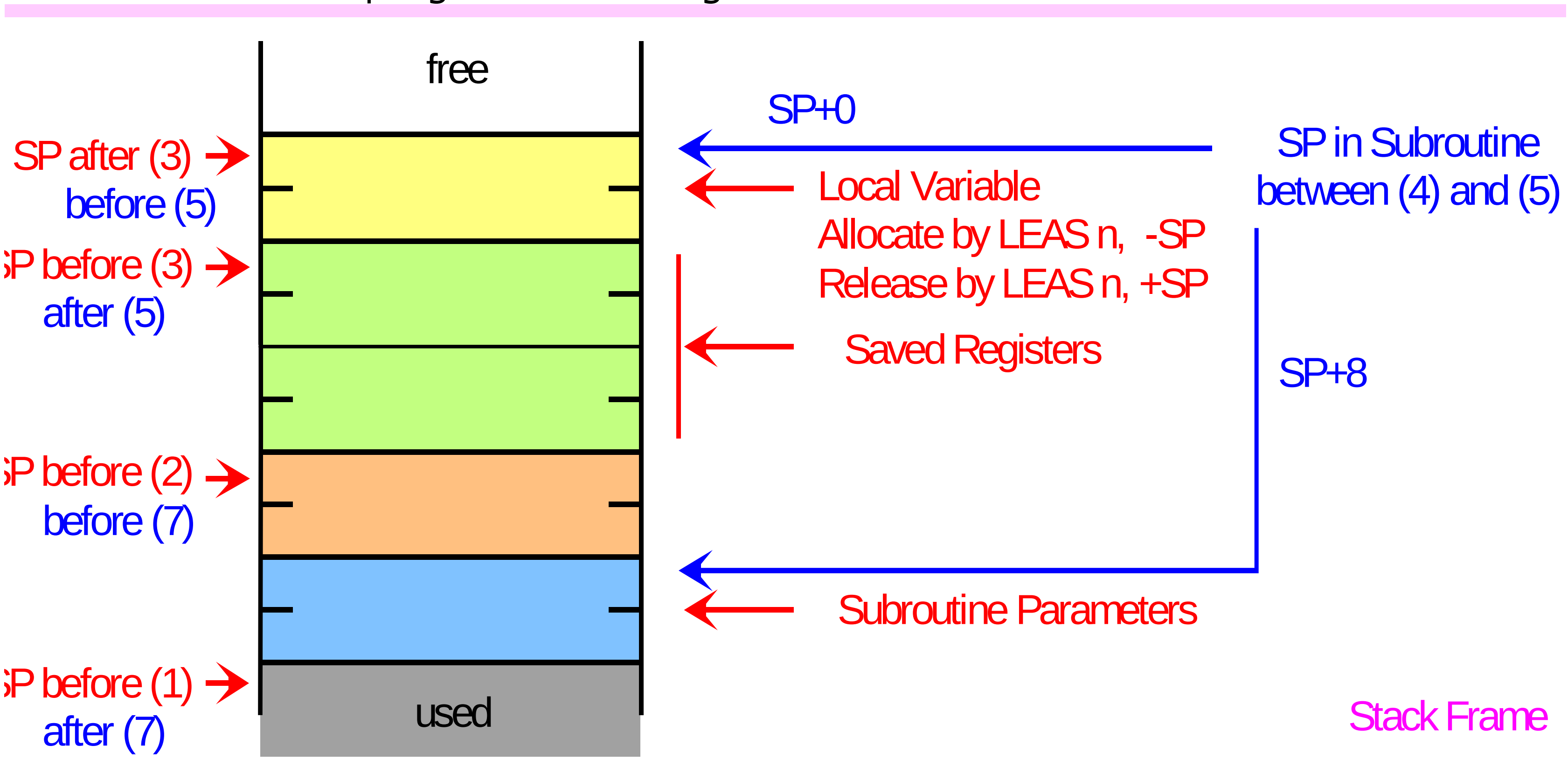


4.4 Local Variables in Subroutines (Stack Frame)

Local variables are considered good style in higher level languages. Unfortunately, assembler does not allow to directly declare local variables, but we can use the same concept which C-compilers use to assign local variables (so-called auto variable) in a so-called **Stack-Frame**:

Example program `LocalVar.mcp` : Search for the maximum in an array → See code on next page

- State of the stack when the program is running



4.4 Stack-Frame

Some C-compilers expect the subroutine to save and restore registers (but never restore the return register!). Not required with CodeWarrior’s HCS12 compiler, but good style.

```
int asmMax (int *pArray, char n);                          // Prototype of assembler function in C
int  val, array[] = { 47, 1600, -4500, 2000, 93, -2010 };  // Array
. . .
val = asmMax(array, 6);                                    // Calling the function          (1)
. . .                    in Assembler: LDD #array, PSHD, LDAB #6, JSR asmMax, STD val, LEAS 2,SP
```

C-Function

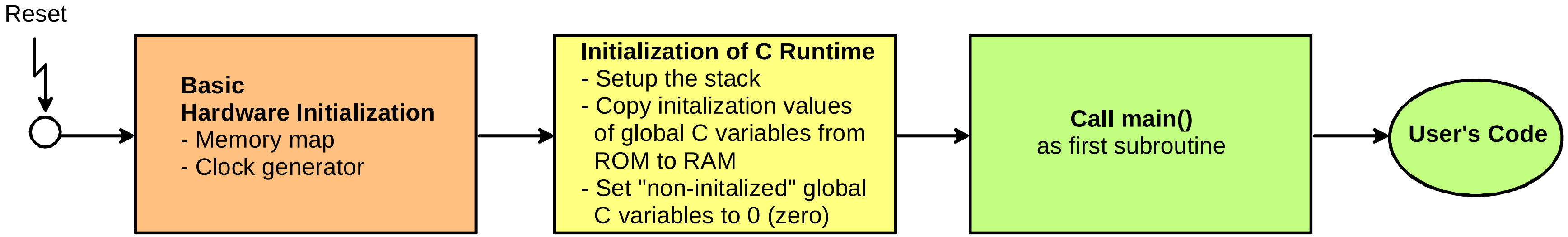
```
int asmMax
        (int *pArray, char n)
{    int max = -32768;



     for (; n > 0; n--)
     {    if (*pArray > max)
          {    max = *pArray;
          }
          pArray++;
     }


     return max;


}
```

Assembler-Function

```
asmMax: PSHX                 ; Save registers                   (2)
        PSHY
        LEAS  2, -SP         ; Allocate stack space ...         (3)
        MOVW  #-32768, 0, SP ;... and initialize local variable (3a)
        LDX   8, SP          ; pArray(from stack)→ X            (4)
                             ; n is in B
for:
        LDY   0, X           ; current array element *pArray → Y
        CPY   0, SP          ; compare it with max
        BLE   next
        STY   0, SP          ; if greater -> max
next:   LEAX  2, +X          ; pArray++
        DECB                 ; n-- for loop counter and loop condition)
endFor: BNE   for            ; if n > 0 go to next element
        LDD   0, SP          ; return max
        LEAS  2, +SP         ; Deallocate space for local variable  (5)
        PULY                 ; Restore registers                (6)
        PULX
        RTS                  ; Return to caller                 (7)
```

4.5 Faster and smaller C/C++ for Embedded Systems

Boot sequence of a stand-alone C-program in an embedded system:



Reset

Basic Hardware Initialization
- Memory map
- Clock generator

Initialization of C Runtime
- Setup the stack
- Copy initalization values of global C variables from ROM to RAM
- Set "non-initalized" global C variables to 0 (zero)

Call main()
as first subroutine

User's Code

More details of the boot process will be discussed in lecture Betriebssysteme [1.5].

Workflow when optimizing a program for size and/or speed

- Define an **architecture** for the program, which is easy to understand and maintain.
- Select the most **efficient algorithms and data types** for the problem (e.g. sort algos)
- Write and debug the program for **correctness first** (not speed or size!).
- Turn on the strongest **optimization** levels **of** the **compiler/linker toolchain**.
- **Profile** the program, to find out which parts are responsible for program size and speed.
- **Optimize the critical parts** using better algos, intrinsics, inline assembler, …

Typical compiler/linker optimizations include

- **Variable placement**, register usage
- **Code modifications**: Loop unrolling or loop-invariant code removal, dead-code removal, …
- "**Smart linkers**" link only those functions of a library, which are actually used.
- . . .

4.5 Embedded C/C++

Compiler and linker optimization

All compiler/linker toolchains can – to a limited extent – optimize programs, but the user must specify the optimization target. E.g. Visual C/C++

Visual C/C++	*GNU C/C++*
cl /O1 myprogram.cpp → **Optimize for code size**	**gcc –Os myprogram.cpp** → **Optimize for size**
cl /O2 myprogram.cpp → **Optimize for speed**	**gcc –O myprogam.cpp** → **Balance size and speed**
. . .	. . .

Optimizations, where the compiler may need the software engineer’s help:

- **Memory** in embedded systems is a limited resource, smaller systems are short of RAM
    - **Constants** should be declared as **const ...** for the linker to place them in ROM.
- The **Stack** should be kept **small**
    - Limit the size of local variables per function, **no large local arrays**
    - **Don’t use recursive algos**, each recursion adds to the stack
    - **Limit the nesting of subroutines**, each subroutine and ISR adds to the stack
    - **Estimate stack usage** in the development phase and **monitor stack usage** during run-time. E.g. periodically check SP, or initalize the stack memory area with a defined byte pattern and periodically check, which part was overwritten, i.e. used.

4.5 Embedded C/C++

- **Inline functions**

  As we have seen in chapter 2.7, calling a subroutine and returning from it is slow, i.e. inefficient for short subroutines. C/C++ allows to declare **inline functions**:

  The programmer writes a normal function, but the compiler decides, whether the function is implemented as subroutine or the function body simply copied into the calling program.

→ Faster code, but code duplicates (memory size), when function is used multiple times.

HCS12 Codewarrior C	*Visual C/C++*

`#pragma inline`
`int strcpy(char *dest, char *src)`
`{ . . .`
`}`
	`__inline int strcpy(char *dest, char *src)`
	`{ . . .`
	`}`
	Hinweis: In C++ auch als **`inline`** **`(ohne __...)`**

- **Register Variables**

  To speed up programs, variables should be held in registers. Better compilers try to optimize register usage automatically. The programmer can help be declaring frequently used local variables with keyword **`register`**, e.g. **`register int a;`**

  The compiler will try to put this variable into a register, if possible. However, many CPUs (HCS12, 80x86, …) have only a small number of registers, so the speed-up may be small.

- **Volatile Variables**

From software’s point of view, hardware registers of peripherals are normal global variables. For all variables, the compiler assumes that only its own code statements will change the value of a variable. Because reading memory is slow, a compiler may hold a local copy of a variable in a CPU register. Thus the compiler may not notice, when the value of a hardware register changes. E.g. in the following program (left column), the compiler reads timer counter register **`TCNT`** only once. To ensure, that always the most current value of a hardware register is used, declare hardware registers with keyword **`volatile`**:

	`short *pTCNT = 0x0044;`	`short volatile *pTCNT = 0x0044;`
`if (*pTCNT > 100)`	`LDD TCNT`	`LDD TCNT` ← 1st read
`. . .`	`CPD #100`	`CPD #100`
	`. . .`	`. . .`
`if (*pTCNT > 200)`	`CPD #200`	`LDD TCNT` ← reread
`. . .`	`. . .`	`CPD #200`
		`. . .`

Compiler intrinsics

Many compilers provide compiler-specific mechanisms to indirectly access low-level features of the CPU. These so called **intrinsics** are implemented as C macros with inline assembler:

HCS12 Codewarrior C	*Visual C/C++*
`EnableInterrupts` → shortcut for **`asm CLI`**	**`_enable()`**
`DisableInterrupts` → shortcut for **`asm SEI`**	**`_disable()`**
`. . .`	`. . .`

- **Bitwise Operations**

Peripherals or low-level communication programs frequently need to modify single bits or groups of bits, without changing other parts of a hardware register. Microprocessors provide special instructions, e.g. BSET, BCLR, BRSET, …. But as there are no equivalent C statements, C programs must use awkward sequences of AND, OR and shift operations.

Example: In PWMPRCLK $x_B=2_D=010_B$ shall be set, other bits must not be changed.

Bit	7	6	5	4	3	2	1	0
Register PWMPRCLK	0	x_B (3bit)			0	x_A (3bit)		

Solution in C:

```
#define PWMPRCLK    (*(char*) 0x00A3)   //Pointer to register
PWMPRCLK = (PWMPRCLK & 0x8F ) | 0x20;
```

The “&” instruction clears bits 6…4. Then the “|” instruction sets bit 5. A non-optimal C compiler does the same, i.e. use AND / OR for byte operands, rather than using the more efficient bit operations BSET / BCLR. The C-programmer must make sure, that constants **0x8F** and **0x50** actually change the desired bits. A better solution is to use a C-**Bit Field**:

```
typedef struct  {   char xA   : 3;       // Definition of a bit field
                    char res3 : 1;
                    char xB   : 3;
                    char res7 : 1;
                } PRCLK;
#define PWMPRCLK (*(PRCLK*) 0x00A3)     // Pointer to register
PWMPRCLK.xB = 2;                        // Set x_B = 2_D
```

4.5 Embedded C/C++

- **Compatibility and Portability**

Assembler programs are **not portable** between CPU families which have a different programming model (register set, instruction set, and address modes). They must be rewritten.

But **low-level C-programs** may **not** be **portable**, too:
Each CPU-family has a **different set of peripherals**. Even if they do the same, e.g. digital I/O, register details are different. This applies even within a CPU family, if it is manufactured by different suppliers, e.g. ARM CPUs of Samsung and Apple have the same CPU instruction set, but have proprietary peripherals.

→ Hardware-dependent parts of programs should be isolated, so that code which needs to be modified, when the CPU family is changed, is easily identified → **HW drivers**.

C has - despite ANSI C89/90 or ISO C99 standards – **gaps in its specification**, i.e. **compilers may not be compatible**. E.g. the size of common data type **`int`** is freely selectable by the compiler supplier. The C standard suggests the size of an **`int`** to be the same as the size of the major data registers and ALU of the CPU, e.g. 16bit for HCS12, 64bit for modern 80x86 (Intel/AMD) CPUs (Note: But Visual C has 32bit **`int`**!!!). In a similar way, data type **`char`** is either **`signed`** or **`unsigned`**, depending on the compiler implementation.

→ Users shall define their own data types in a central header file and never use the original C data types directly, e.g.

```
typedef  unsigned char  uint8_t
typedef  signed char    int8_t
typedef  unsigned int   uint16_t
```

When using another compiler, only the central header file needs to be adapted.
(C99 compatible compilers provide such a header file named “**`stdint.h`**”).

Most compilers provide **proprietary language extensions**. E.g. HCS12 CodeWarrior C:

The C standard says, to access compiler-specific extensions the programmer shall use keyword **`#pragma`**, e.g.

```
#pragma TRAP_PROC    // Defines the following subroutine as ISR
void myInterruptServiceRoutine(void) { . . . }
```

CodeWarrior's proprietary solution requires only a single line of code. However, its non-standard keyword **`interrupt`** may not be understood by other C compilers:

```
void interrupt 23 myInterruptServiceRoutine(void) { . . . }
```

Intrinsics (see above) typically are compiler-specific, i.e. not portable.

Memory addresses can be accessed in a C-standard conformant way via pointers, e.g.

```
char *p = 0x0001;   //Assign PORTB's address to pointer
*p = *p | 0x02;     //Set Bit 1 in Port B
```

or via a macro

```
#define PORTB (*((char *) 0x001))
PORTB = PORTB | 0x02;
```

or non-standardized via

```
char PORTB @0x0001;    //Assign address to variable
PORTB = PORTB | 0x02;  //Set Bit 1 in Port B
```

Note: The **same compatibility issues occur with other programming languages**, e.g. Android's Java SDK (Software Developer Kit) is not compatible with Sun/Oracles Java SDK.

- **Execution Speed depends on Data Types**

Speed in CPU clock cycles, 1 clock = 42ns @ f_{BUSCLK}=24MHz, measured with the HCS12 True Time Simulator, C-code with compiler default optimization, all operands and results are global variables, execution clock cycles include the arithmetic instruction plus fetching the operands and storing the results from/to memory.

Operation		*Data type of operands a, b, c*			
		int *(16bit)*	*long* *(32bit)*	*float* *(IEEE 32bit)*	*double* *(IEEE 64bit)*
Addition/subtraction	c = a + b	9	21	203	500
Multiplication	c = a * b	12	83	285	4174
Division	a = c / b	21	143	1049	5324
Type conversion from **`int a`** to . . .		**`b=(int)a`** 6	**`b=(long)a`** 22	**`b=(float)a`** 127	**`b=(double)a`** 969

Analysis of the results:

- Add and subtract is fast, multiply is slower (if a CPU does not have a full operand-width hardware multiplication unit), divide is a disaster
- Data types, which are longer than the size of the ALU (16bit @ HCS12) lead to slow execution.
- Integer arithmetic is much faster than floating-point arithmetic (if the CPU does not have a floating-point ALU)

→ Use integer arithmetic and smallest-size data types, but check overflow and resolution

→ Use efficient algorithms, i.e. algorithms with the fewest number of multiply operations and no divisions, if possible.

C++

- Has a bad reputation with respect to code size and speed, especially in embedded systems:
  - **C++** compilers are much **more complex than C** compilers
  - **Some vendors don't spend much development effort on** the **C++** features of their compilers, especially if the compiler is for free and/or has no large user community.

  → Use a compiler with a large sales volume or a costly professional compiler (e.g. Gnu C → Microsoft Visual C++ → Intel C++)

- Engineers need to know, which language features do cause overhead and which don't: (see e.g. Scott Meyer: Effective C++, Addison-Wesley Publishers)

  Nearly no runtime overhead or code bloat, because the following **C++ features** are **resolved** by the compiler **during compile-time**:

  - **C++ class attributes** are **implemented in the same way as C structure variables**, but with two additional function calls (constructor and destructor), when the class is instantiated. However, same code required when a C structure is initialized.
  - **C++ class methods** are **implemented like C functions with one additional parameter** (a `this` pointer to the associated object)
  - **Simple inheritance of non-virtual classes, default arguments** for functions, **namespaces**, **overloaded functions and operators**, **private/protected/public declarations** only specify the scope of variables and methods during compilation.
  - C++ **classes and structures** as function parameters should be **passed by** (a **const**) **reference** (= pointer), not by value (=copy of the class/structure).

- **Dynamic creation of** C++ **objects** via **new/delete** may lead to the same problems like dynamic C data structures created via **malloc()/free()** → heap fragmentation, memory leaks, out of heap memory problems

- C++ uses much **stricter rules for data type compatibility** and requires many explicit type casts rather than the more relaxed automatic casts in C. **To allow type checks when linking** code modules, compilers "decorate" C++ function and variable names in the object code with data type information ("**name mangling**"), e.g. if the programmer defines a function `int fcn(char x)`

  Visual C++ may decorate the name as `?fcn@@YAHD@Z`

  where the `@...'s` indicate the type of the parameter and the return value. Unfortunately, name mangling is compiler specific and only done in C++. If this function is called from C or from an assembler module, which do not use mangling, the linker will stop with an error, because the function names between the C++ and the non-C++ module do not match. A cross-language compatible C++ function must be declared as

  ```
  extern "C" int fcn(char x)
  ```

  to turn off name mangling for this function. Mangling is done in the compile-link phase, it has no influence on code size or runtime execution speed.

C++ features with negative effect on size or speed (when actually used):

- **Inheritance from a virtual base class** causes an array of pointers attached to the class (**Virtual Function Table VTABLE**) → memory size ↑. The table is used to indirectly call the function implementations via the pointers → runtime ↑.
- **Exception handling** with **`try {} catch {}`** stores the current state of the program (CPU registers etc.) on the stack, before the **`try {}`** block is executed → stack size↑. If an exception occurs inside the **`try`** block, this state is restored to recover from the exception → runtime ↑. As in normal program flow no exception should occur, compilers try to optimize the **`try {}`** path at the expense of the **`catch {}`** path.
- **"Expensive" library functions**: Some standard library functions like **`printf`** or **`iostream`** have to take care about any possible parameter combination and thus require a lot of code. For embedded applications there are simplified versions of the standard libraries, e.g. **`uClibc`** instead of **`glibc`** for GNU compilers.
- **Templates** may or may not increase code size, depending on how complex and nested the generated classes are.
- **Runtime-Type Identification RTTI** (**`dynamic_cast`**) allows to call a function with different object types, not known during compile time. This is one of the most esoteric C++ features and thus, compiler implementations may be very inefficient. To be avoided.

Chapter 5

Advanced Microprocessor Architectures Intel x86 - ARM

5.1 Overview

Microprocessors for General Purpose Computers

CPU type	*Supplier*	*Type/Data width*	*Comment*
• **PCs, Workstations, Notebooks**			
80x86, x86	Intel, AMD	CISC, 32/64 bit	Brand names: Core, Pentium, Atom, Athlon, Opteron, Ryzen, …
~~Power PC~~	~~IBM, Freescale~~	~~RISC, 32/64 bit~~	~~Till 2005 used in Apple Macintosh computers~~
• **Server** (file server, web server, database server)			
80x86	See above		
~~Itanium~~	~~Intel (HP)~~	~~RISC, 64bit~~	~~Dead end, development stopped~~
Sparc	Sun	RISC, 64bit	2009 Sun bought by Oracle
Power	IBM	RISC, 64bit	Upward compatible to Power PC
• **Mainframes** for commercial software and large technical computation (vector computers, number crunchers)			
Proprietary and x86-Clusters	IBM, Cray, SGI, NEC, Fujitsu		IBM Blue Gene with Power, Cray XT with AMD Opteron/Ryzen,…
Nvidia GPU	Nvidia	Multi-core RISC	Massive parallel processing for Machine Learning

Microprocessors for Mobile Devices

• **Game Consoles**			
Power	IBM	RISC, 64bit	Microsoft XBOX 360, Nintendo Wii/Game Cube
~~Cell~~	~~IBM~~ ~~(Sony, Toshiba)~~	~~RISC, 64bit~~ ~~Extreme Multi-Core-CPU~~	~~Sony Playstation 3 (combined with Power PC CPU for management tasks)~~
x86	Intel, AMD	CISC 32/64bit	Microsoft XBOX One, Sony Playstation 4
• **Smartphones, Tablets**			
ARM 9, ARM 11 ARM Cortex M ARM A…	Miscellaneous (Apple, Samsung, Qualcomm, …)	RISC, 32bit/64bit	Qualcomm, Samsung, Apple, NXP/Freescale ... with Digital-Signal-Processor DSP and Java extensions
80x86	Intel	CISC 32bit/64bit	Brand name: Atom for Windows and Android!

5.1 Overview

Microcontrollers for Embedded Systems

CPU type	*Supplier*	*Type/Data width*	*Remark*
8051	Miscellaneous (Origin: Intel)	CISC, 8bit	Licensed and enhanced by many suppliers, e.g. Philips/NXP, Siemens/Infineon, …
680x	NXP+ Freescale/Motorola	CISC, 8bit	CPU families 6805, 6808, 6809
681x		CISC, 16bit	CPU families 6811, 6812, 6816
68xxx		CISC, 32bit	CPU families 68000 – 68060, 68332, ColdFire, …
MPC55xx		RISC, 32bit	Embedded Power PC MPC555, 5556, …
PIC 1x PIC 24	Microchip (Atmel)	RISC, 8bit RISC, 16bit	Families 12, 14, 16, 18 (= size of CPU address)
AVR 8 AVR 32		RISC, 8bit RISC, 32bit	ATtiny, ATmega
C16x TriCore TC1xxx	Infineon (Siemens)	RISC, 16bit RISC, 32bit	Used in many automotive ECUs
R8C, M16C R32C, SuperH 78Kxx V850	Renesas (Joint venture of Hitachi, Mitsubishi and NEC)	 CISC, 8/16bit RISC, 32bit	Many different product families from 4bit to 32bit, collection of the proprietary CPUs of the mother companies of the joint venture
MIPS	Imagination (MIPS Technology)	RISC, 32/64bit	Originally for workstations/servers, today used in routers and settop boxes, e.g. AVM Fritz Box
ARM 7, ARM 9 ARM Cortex M/R	Miscellaneous	RISC, 32bit	Licensed to many suppliers, e.g. NXP/Freescale, Atmel, Philips, STM, Apple (!), Microsoft (!), …

Very broad market, dominated by many „old“ architectures (8051, 68xx developed in the 70s/early 80s), large range of CPU performance, memory size and peripherals, strongest growth in recent years: ARM

5.2.1 Power Saving Mechanisms

Purpose: Reduce cooling effort

Increase battery operating time

Idea: Electrical power loss for CMOS logic $\mathbf{P \sim U^2 \cdot f}$

But: CMOS transistors switch faster at higher gate voltages

→ lower supply voltage U only possible at lower clock frequency f

Measures: Reduce supply voltage U and clock frequency f, if no/low computing performance is required

→ Programmable PLL for clock generator

→ Programmable voltage supply

Switching occurs in steps:

- Switch off peripherals
- Switch of (part of) the bus system
- Switch of CPU → reactivated periodically by timer interrupts or if required by an interrupt of a peripheral, i.e. a key press or an incoming network message

5.2.2 Protection and Monitoring

Privilege Levels and Privileged Instructions

Purpose: Operating system shall monitor user programs and check their resource usage (CPU time, memory)

Idea: Operating system monitoring periodically activated by timer interrupt

Measures: CPU switches between two privilege levels

- Kernel/system mode → operating system
- User mode → user programs

Critical instructions may only be executed in kernel mode:

- Enable/disable interrupts, i.e. user programs cannot disable interrupts (and thus cannot stop the operating system monitoring)
- Write (and read) peripheral registers and other management structures, i.e. the interrupt vector table
- User programs may only call operating system services via a protected mechanism, i.e. software interrupts

5.2 CPU Support for Operating Systems

Run-time Monitoring

Purpose: CPU shall be stopped/restarted, if the CPU “hangs”/runs wild due to a software bug

Idea: Monitor, if a certain task is executed correctly and in time periodically

Measures: Hardware timer (**Watchdog**), which must be written periodically, i.e. by writing certain data values in a register

If the watchdog is not written correctly, the watchdog will reset the CPU

Detection of Memory Errors

Purpose: Detection of „flipped“ memory bits (reliability problem of large Dynamic RAMs or Flash ROMs at high temperatures)

Idea: Store redundant parity bits/check sums (CRC), generated by a hardware unit when writing data and check them when reading. Stop program on error or automatic error correction (ECC Error Checking and Correction)

Detection of CPU Hardware Errors

Idea: Program executed on two identical CPUs clock by clock (**Lockstep CPU**) → very costly, used in safety critical systems

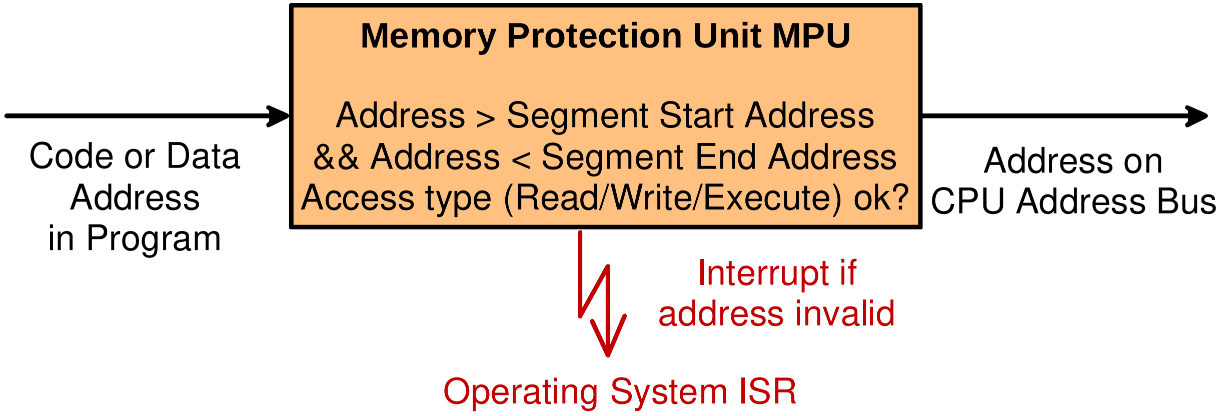
Measures: Results of ALU operations etc. monitored by hardware comparators, reset the CPUs if results do not match

Memory Protection (Segmentation)

Purpose: A user program shall not access the memory area of the operating system or of other user programs

Idea: Each program gets its an exclusive memory address range (Segment)

Measures: Hardware monitoring and triggering interrupts, if a program tries to access a memory address (code or data) in a foreign memory address range


Memory Protection Unit MPU
Address > Segment Start Address
&& Address < Segment End Address
Access type (Read/Write/Execute) ok?
Code or Data Address in Program
Address on CPU Address Bus
Interrupt if address invalid
Operating System ISR

Example Freescale HCS12X

- 2 operating modes (supervisor and user state)
- 8 memory segments with configurable start and end addresses plus access rights (read, write and/or execute)

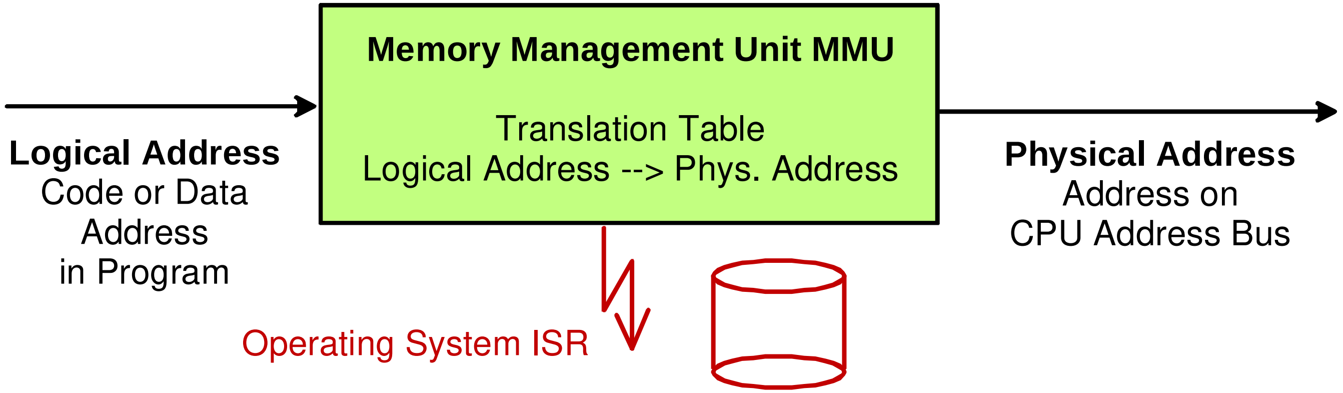
5.2.3 Memory Extension (Virtual Memory / Paging / Swapping)

Purpose: Programs shall be able to use more memory than physically available or more than the CPU’s address range allows

Idea: If the memory is larger than the CPU’s address range (i.e. HCS12 DP256 with 256KB memory despite of 16bit addresses only): Switching of memory blocks (**Pages**) via programmable address decoders

If more memory required than physically available: Extend the memory by copying currently unused data/code to the harddisk (**Swapping**)

Measures: Swapping shall work independent from the user program (transparent): Conversion of memory addresses via an address translation table (**Page Table**) plus operating system interrupt services



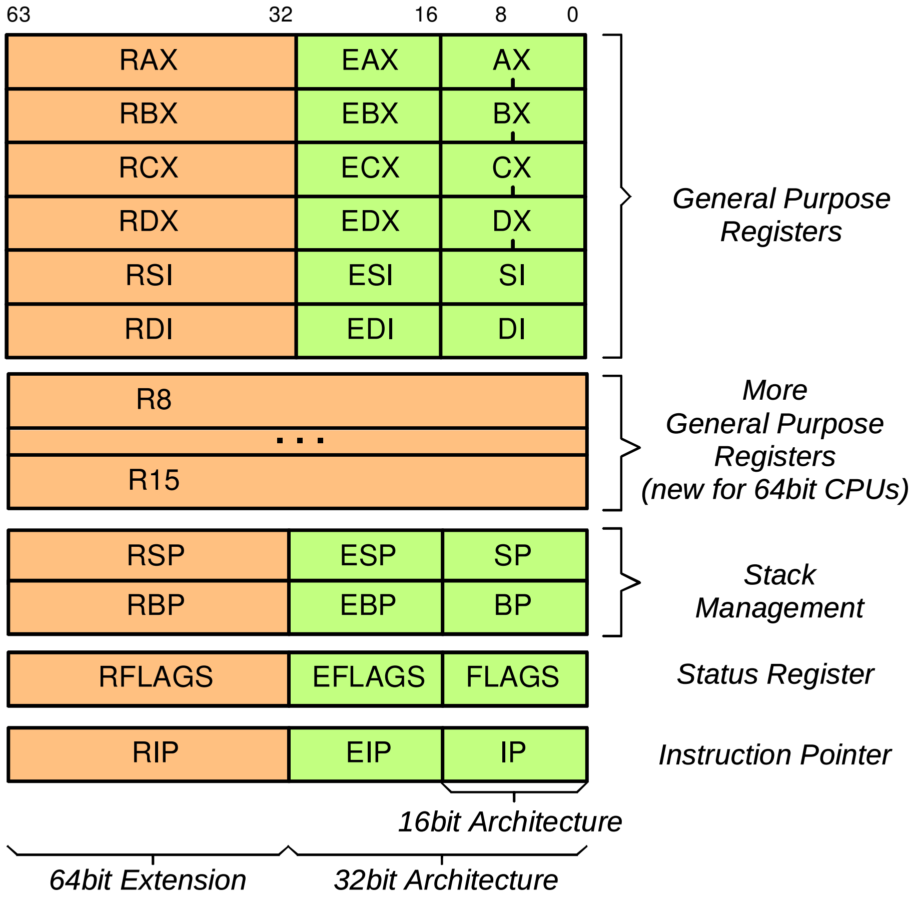
If no memory space is free: Copy currently unused memory content to hard disks
If logical address currently not in memory: Copy from harddisk to memory

80x86-Architecture

(see http://developer.intel.com)

- **Programming Modell** for User Programs

Register Set



63 … 32	32 … 16	16 … 0
RAX	EAX	AX
RBX	EBX	BX
RCX	ECX	CX
RDX	EDX	DX
RSI	ESI	SI
RDI	EDI	DI
R8		
. . .		
R15		
RSP	ESP	SP
RBP	EBP	BP
RFLAGS	EFLAGS	FLAGS
RIP	EIP	IP

Floating point registers (FPU, SSE) not shown.

History

- 8bit architecture 8080, 8085 (1974)

Since 1979 upward compatible CPU families

- 16bit architecture 8086, 80186, 80286
- 32bit architecture 80386, 80486, Pentium
- 64bit architecture AMD64,Core EM64, i3/5/7

Several Operating Modes

- Real Mode 16bit DOS (IBM PC 1981)
- Protected Mode 32bit Windows (Win32)
- Compatibility mode 16bit/32bit (Win16)
- 64bit Mode
- Compatibility mode 32/64bit (Win64)

Characteristics

- Small register set (only 6 general purpose 32bit regs.)
- **CISC** instruction set >> 100 instructions
- **Von-Neumann** memory model
- **Little Endian**, byte addressable
- Address- and data size 32bit/64bit
- **2 Address Instructions**, 1 register operand and 1 register or memory operand

5.3 PC and Server-CPUs Intel/AMD 80x86

Example of 80x86 assembler instructions:

```
.data
myVar DD 01234567h, 89ABCDEFh          Definition of an array with two 32bit elements
.code
MOV ESI, 4                             ESI = 4
MOV EAX, 1                             EAX = 1
ADD EAX, myVar[ESI]                    EAX = EAX + myVar[ESI] = 1 + 89ABCDEFh
```

Possible address modes

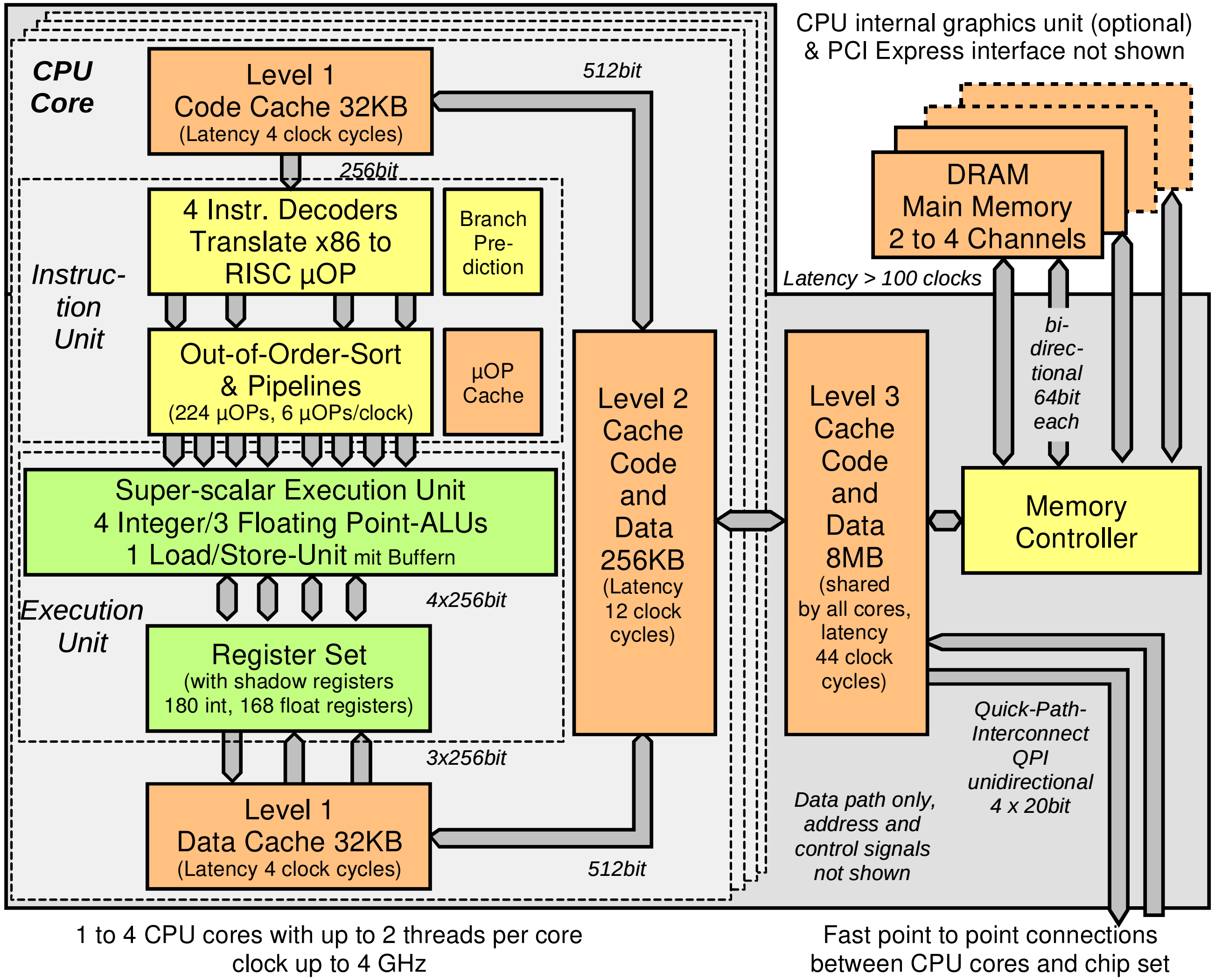
- register addressing
- immediate addressing
- direct, register-indirect and indexed memory addressing
- Single Instruction Multiple Data (SIMD) instructions with up to 4 x 2 32bit operands
- Floating-point instructions with 32bit, 64bit and 80bit floating-point operands

Orthogonal instruction set

- Each register can be source or destination operand in (nearly) all instructions

Major differences to HCS12	*Intel*	*Motorola/Freescale*
• Memory order	Little Endian	Big Endian
• Operand order	Destination – Source i.e. **MOV EBX,EAX EBX ← EAX**	Source – Destination **i.e. TFR A,B A → B**
• Constants	**MOV EAX, 1**	**LDAA #1**
• Hexadecimal values	**01234567h**	**$01234567**

Microarchitecture of current 80x86-CPUs (Core i7-6xxx SkyLake 14nm, i7-7xxx KabyLake 14nm)

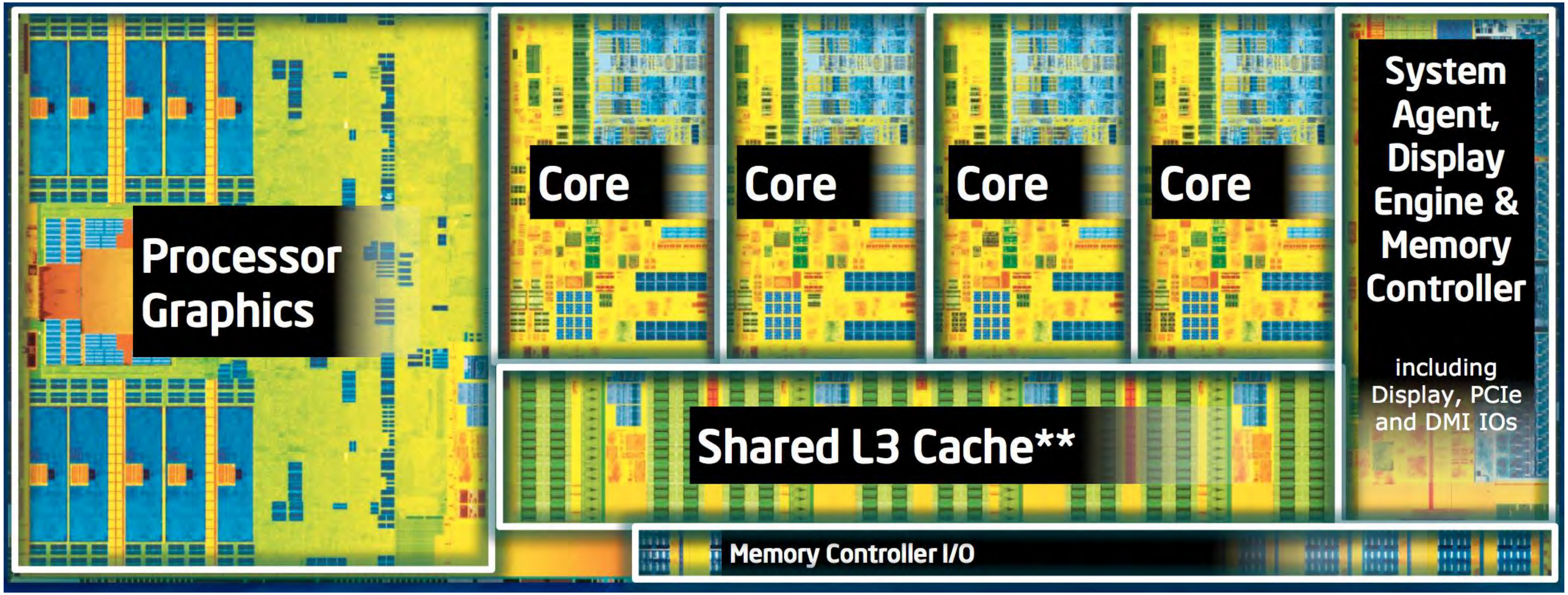

CPU Core
Level 1 Code Cache 32KB (Latency 4 clock cycles)
512bit
256bit
4 Instr. Decoders Translate x86 to RISC µOP
Branch Pre-diction
Instruc-tion Unit
Out-of-Order-Sort & Pipelines (224 µOPs, 6 µOPs/clock)
µOP Cache
Super-scalar Execution Unit 4 Integer/3 Floating Point-ALUs 1 Load/Store-Unit mit Buffern
Execution Unit
4x256bit
Register Set (with shadow registers 180 int, 168 float registers)
3x256bit
Level 1 Data Cache 32KB (Latency 4 clock cycles)
512bit
Level 2 Cache Code and Data 256KB (Latency 12 clock cycles)
Level 3 Cache Code and Data 8MB (shared by all cores, latency 44 clock cycles)
CPU internal graphics unit (optional) & PCI Express interface not shown
DRAM Main Memory 2 to 4 Channels
Latency > 100 clocks
bi-direc-tional 64bit each
Memory Controller
Quick-Path-Interconnect QPI unidirectional 4 x 20bit
Data path only, address and control signals not shown
1 to 4 CPU cores with up to 2 threads per core clock up to 4 GHz
Fast point to point connections between CPU cores and chip set

The current **microarchitecture** is much more complex than the 30 years old programming model:

- **Internal** memory in **Harvard** structure, **external** memory still **Von-Neumann**
- 80x86-**CISC** instructions **internally converted** to **RISC** instructions via 4 instruction decoders
- **Superscalar execution unit** (Multiple Issue with max. 6 parallel RISC instructions)
- **Medium sized pipeline** with (approx.) 16 stages
- **Out-of-Order** execution with a large set of **shadow registers**
- Dynamic **clock switching**

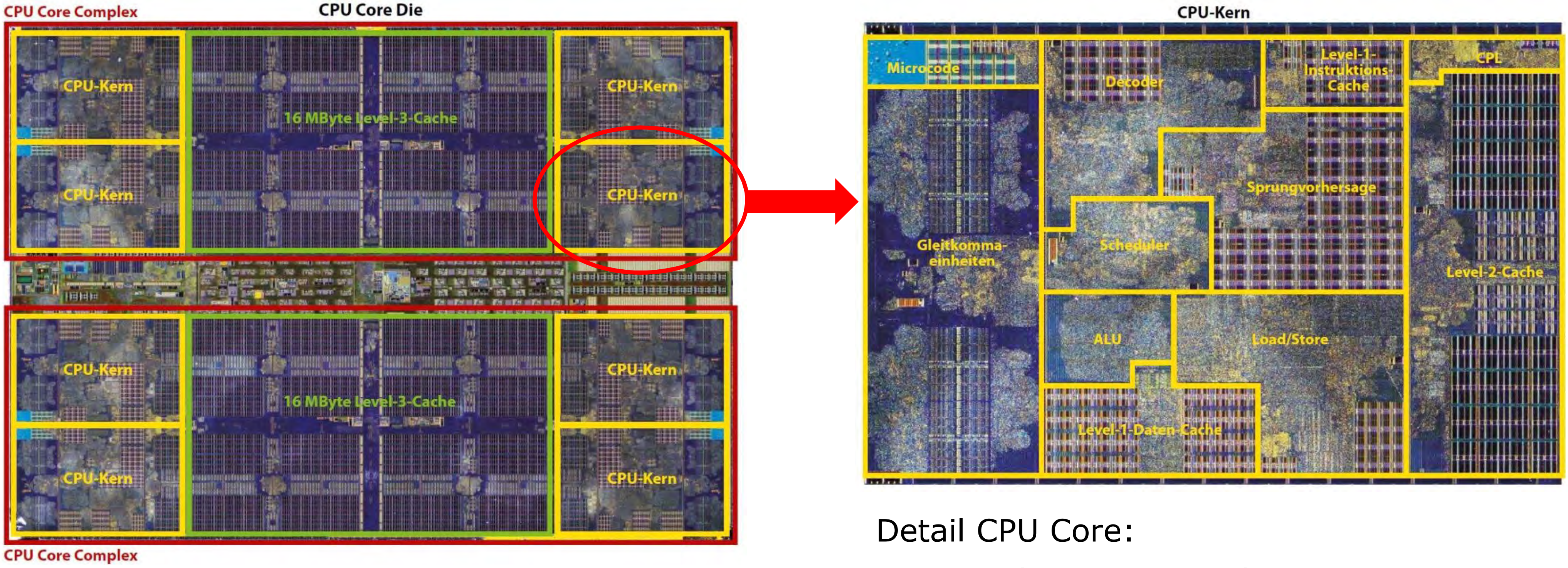
Chip Photo **Intel Core i7**

4 CPU cores with common 8 MB level 3 cache and GPU (Graphic Processing Unit) on chip


Processor Graphics
Core
Core
Core
Core
System Agent, Display Engine & Memory Controller
including Display, PCIe and DMI IOs
Shared L3 Cache**
Memory Controller I/O

Source: http://www.anandtech.com/show/7003/the-haswell-review-intel-core-i74770k-i54560k-tested

Chip Photo **AMD Ryzen 3000**: Multi-Chip-Design with CPU Core Die and I/O Die (Chiplets), separate GPU



CPU Core Die:

2 x 4 CPU cores
2 x 16 MB Level 3 cache
74 mm² chip size @ 7 nm technology
3.9 billion transistors

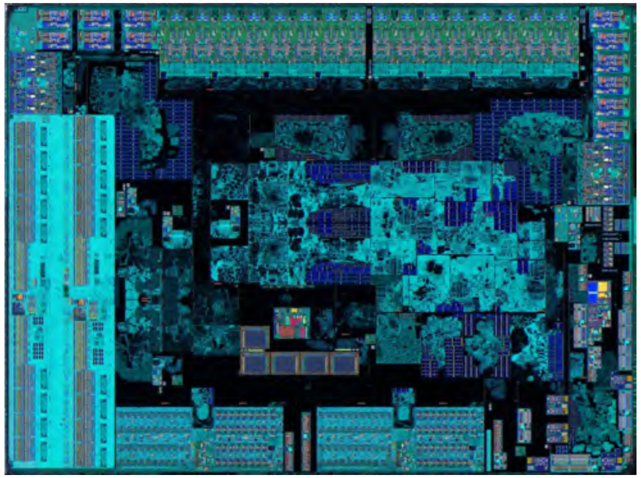
Detail CPU Core:

512 kB Level 2 cache,
32 kB Level 1 Instruction cache
32 kB Level 1 Data cache
4 x 64 bit Integer ALUs
4 x 256 bit Floating point ALUs (ADD/MUL)

Superscalar out of order execution decoder/scheduler with μOPs
Pipelining with branch prediction unit
Register file / operand handling in load/store unit
Clock, power & temperature monitoring (CPL)

5.3 PC and Server-CPUs Intel/AMD 80x86

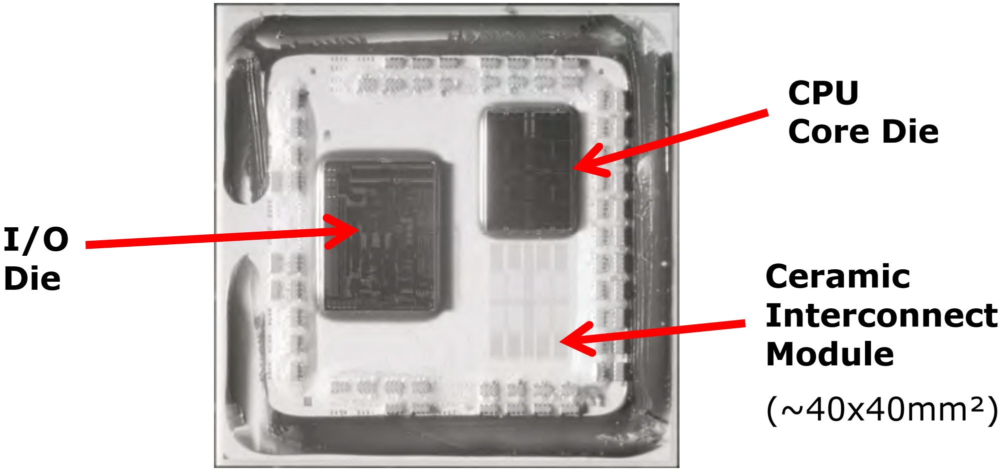
AMD Ryzen 3000 continued



I/O Die:

Memory controller, PCIe, SATA, USB
125mm² chip size @ 12 nm technology
2.1 billion transistors

Multi-Chip-Design
(Module with heat spreader removed)



CPU Core Die

I/O Die

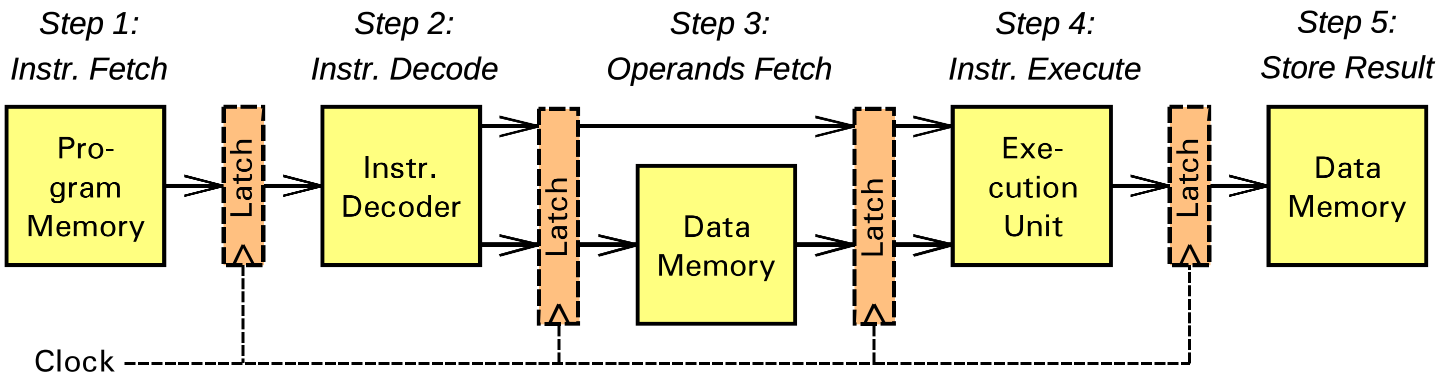
Ceramic Interconnect Module

(~40x40mm²)

Variants with several CPU dies and/or additional GPU modules (not shown)

Sources:
Christian Hirsch: Grundlagen zu Prozessoren. c't 11/2020, pg. 140-141, heise.de
https://www.amd.com/de/ryzen
https://www.hardwareluxx.de/index.php/galerie/komponenten/prozessoren/amd-nexthorizone3-mikeclark.html
https://www.tweakpc.de/hardware/tests/cpu/amd_ryzen_7_3700x_ryzen_9_3900x/s02.php

Pipelining: Interleaved execution of instructions (*Instruction Level Parallelism ILP 1*)



Program Memory	Instruction Decoder	Data Memory	Execution Unit	Data Memory	↓ t
Fetch instruction 1	. . .	. . .	. . .	. . .	1
Fetch instruction 2	**Decode instruct 1**	. . .	. . .	. . .	2
Fetch instruction 3	*Decode instruction 2*	**Fetch operands 1**	. . .	. . .	3
Fetch instruction 4	Decode instruction 3	*Fetch operands 2*	**Execute instruct 1**	. . .	4
Fetch instruction 5	Decode instruction 4	Fetch operands 3	*Execute instruction 2*	**Store result 1**	5
Fetch instruction 6	**Decode instruct 5**	Fetch operands 4	Execute instruction 3	*Store result 2*	6
. . .	. . .	. . .	. . .	. . .	...

Precondition: All instructions use all 5 stages → add NOP steps if needed
Slowest stage defines clock period

Latency: Time from Fetch Instruction X to Store Result of X → 5 clock periods

Throughput: 1 instruction per clock period (without pipelining: 1 instruction per 5 clock period)
→ Theoretical speedup x n with n pipeline stages

Problems with Pipelines (**Pipeline Hazards**)

- Data bus conflict

  Example in clock period 5: Fetch instruction 5 – Fetch operands 2 – Store result 1

  → Implement separate busses for instructions, operands and results

- Next instruction not always known

  Example: Conditional branch instruction, when the immediately preceding instruction calculates the condition

  → Branch prediction required to avoid pipeline stall/flush

- Data dependency

  Example in clock period 5: If result of instruction 1 is operand of instruction 3

  → Bypass the store result cycle and forward result 1 directly to the execution unit

  Example: If result of instruction 2 is operand of instruction 3

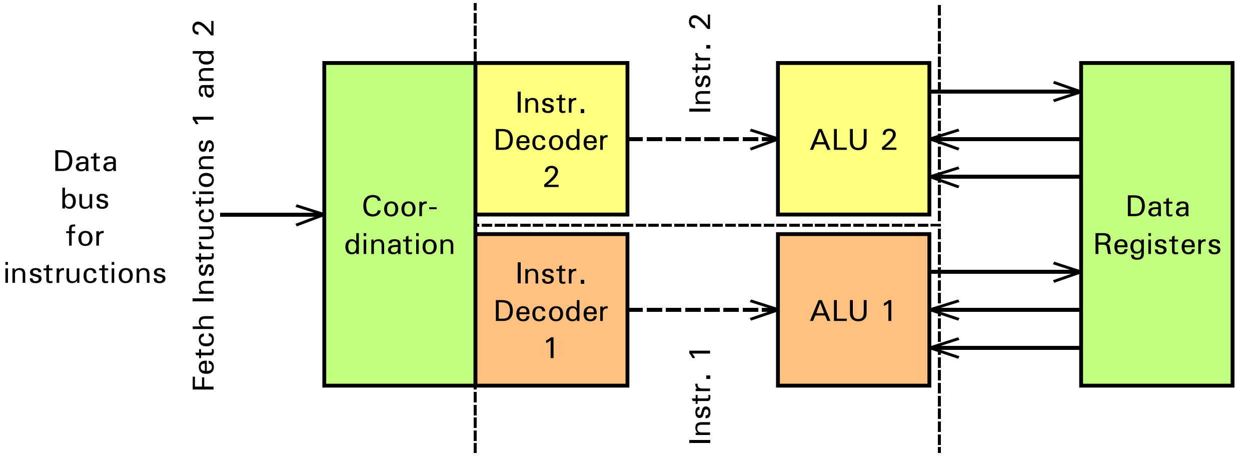
  → Reorder instruction sequence

Superscalar CPU: Parallel Execution of Instructions

(***Instruction Level Parallelism ILP 2: Multiple Issue***)

- Parallel exection of multiple instructions in multiple execution units (ALUs)

  Example with 2 ALUs:



Throughput: Theoretical speedup x m with m execution units

Problems

- Data bus conflict → same as pipelining
- **Data dependency between parallel instructions** → same as pipelining

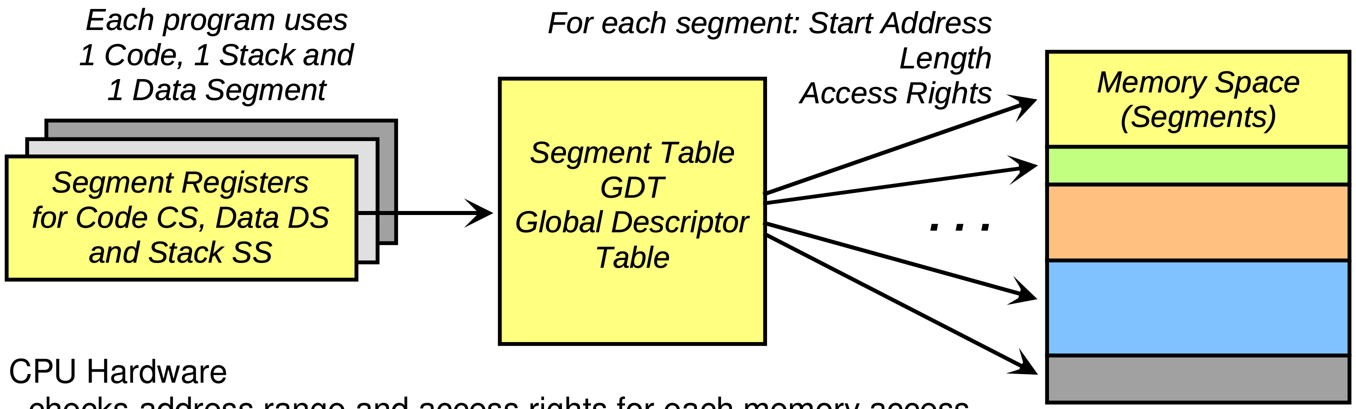
  Example: Instructions R1+1→ R2, R3+2→ R4 → no problem with parallel execution

  R1+1→ R2, R2+2→ R4 → 2nd needs result of 1st instruction

  → Change order of instruction (Out of Order Execution by compiler or by hardware)

CPU Protection Methods for Operating Systems

- Privilege levels
  - Each program uses one of two privilege levels: *Kernel Mode – User Mode.* The level assignment is done via *segment registers* for code, data and stack and via a *segment table*
  - Operating systems run in *Kernel Mode*, user programs run in *User Mode*
  - *User Mode* programs cannot access *Kernel Mode* data
  - *User Mode* programs can call *Kernel Mode* functions only via predefined addresses (*Call Gates*)
- Privileged instructions
  - Critical instructions, i.e. enable/disable interrupts, input/output instructions to access peripherals or reading/writing registers of the memory management unit, are only allowed in *Kernel Mode*.
- Memory management and access →Memory Protection MPU and Memory Management MMU



Each program uses 1 Code, 1 Stack and 1 Data Segment

Segment Registers for Code CS, Data DS and Stack SS

Segment Table GDT Global Descriptor Table

For each segment: Start Address Length Access Rights

Memory Space (Segments)

. . .

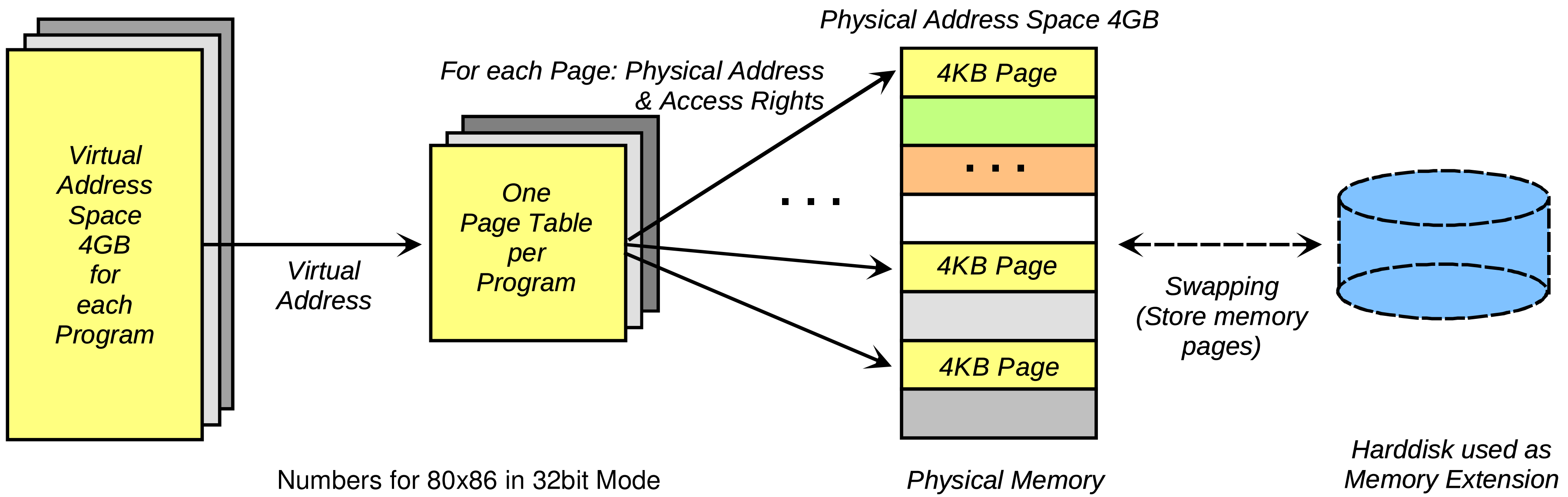
CPU Hardware
- checks address range and access rights for each memory access
- interrupts program (Exception Interrupt), if access invalid

Memory protection by **Segmentation**

- Divide memory into variable-length areas (segments)
- Access rights for each segment:
  Code – Data
  Kernel – User Mode
  Read–Write–Execute

Memory Management by **Paging**

- Each program virtually holds the full 4GB (in 32bit mode) address range
- Its memory range is divided into small, fixed sized areas, i.e. 4KB (*Pages)*
- All virtual pages are mapped to real physical memory via page tables
- Each program has its own page table which only contains the program's own pages, i.e. a program sees only its own memory, not the pages of other programs
- If a program runs out of physical memory, physical memory pages (of the same or of other programs), which currently are not used, are copied to the harddisk (*Swapping*)
- Memory pages have access rights. If a program tries to access a page, for which it does not have the correct access rights, an *Exception Interrupt* will be triggered. The operating system ISR then decides, if the program is stopped, if it gets a new page (i.e. if the stack overflows and is automatically extended) or if the missing page is fetched from the harddisk, if it was swapped before.


Physical Address Space 4GB
For each Page: Physical Address & Access Rights
Virtual Address Space 4GB for each Program
Virtual Address
One Page Table per Program
. . .
4KB Page
. . .
4KB Page
4KB Page
Swapping (Store memory pages)
Harddisk used as Memory Extension
Physical Memory
Numbers for 80x86 in 32bit Mode

5.3 PC and Server-CPUs Intel/AMD 80x86

Segmentation (Memory Protection) and Paging (Memory Management) are alternatives for memory management and protection. Segmentation is simpler to implement than paging.

80x86-CPUs implement both, but operating systems typically concentrate on one of the two methods. E.g. DOS/Windows 3.1 used segmentation, Windows XP/Vista/7/8/10 and Linux use paging.

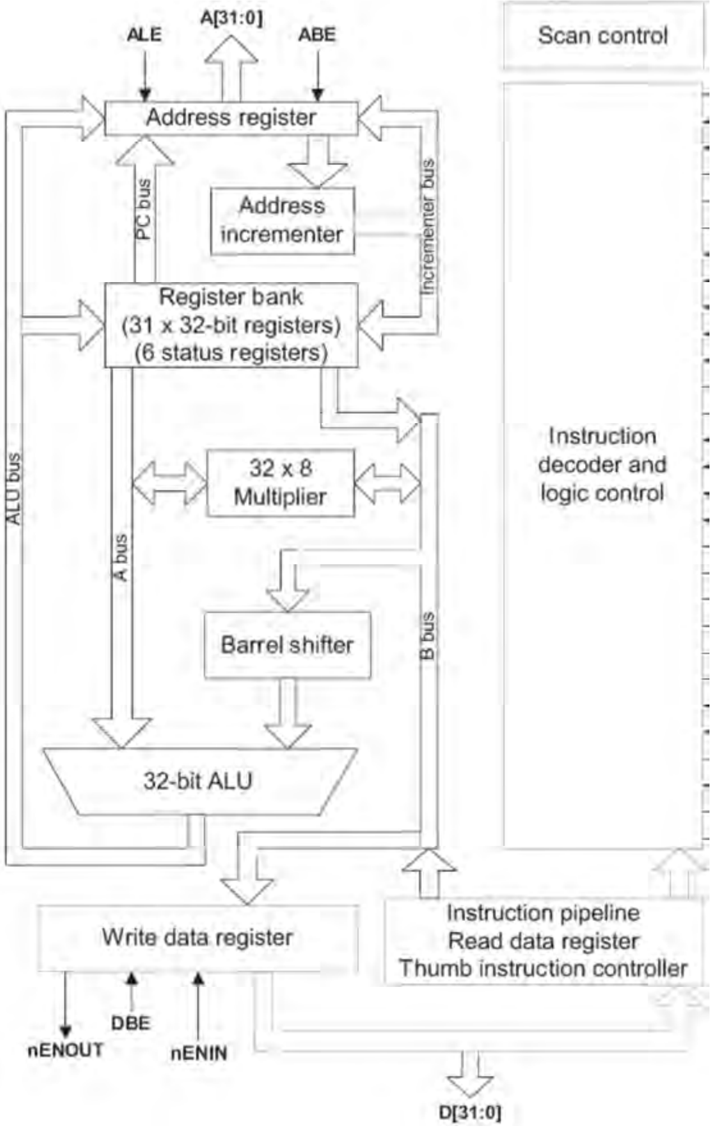
Outlook: When will PC CPUs run out of memory again?
This happened with 32bit addresses by end of the last decade and with 24bit addresses beginning of the 1990s.

- Today's 80x86-CPUs theoretically have 64bit addresses, but only 40 (48) address lines are available as pins. When will this memory limit be reached?
- Moore's "Law":
  Number of transistors per chip (= number of bits for ROMs and DRAMs) doubles at (nearly) constant cost every 2 …3 years
- Starting point:

2014	32 bit addresses	4 GByte (standard for 500€ PCs)
2016	33 bit	8 GByte
2018	34 bit	16 GByte
2020	35 bit	32 GByte
. . .	. . .	. . .
2030	40 bit	1 TByte
. . .	. . .	. . .
2046	48 bit	256 TByte

ARM-32bit-Architecture

(see http://www.arm.com)

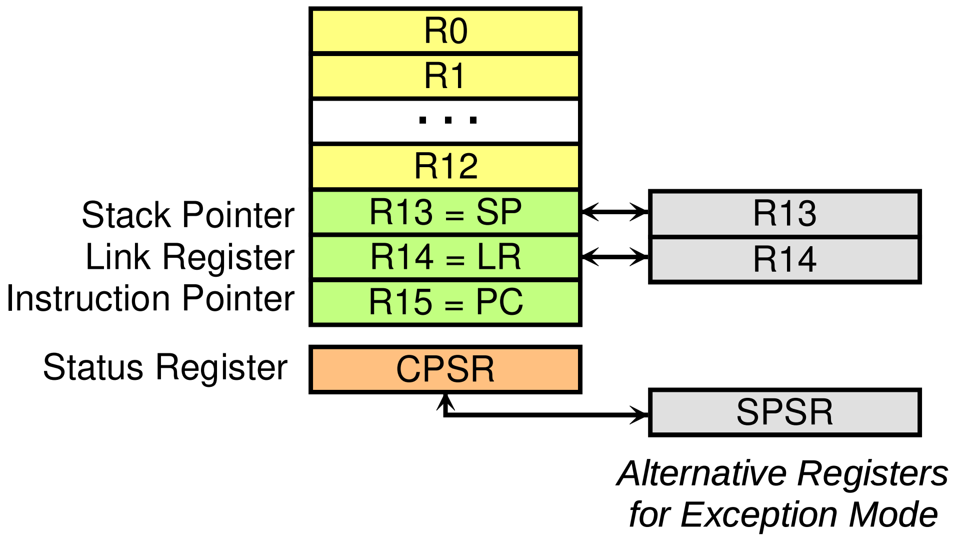


ARM 7

- Typical **32bit RISC** architecture (ARM7/9/11, Cortex Mx)**, extended to 64bit** (ARM Ax)
- Family of upwards compatible CPUs from simple microcontrollers without operating system to microprocessors for mobile computers
- Tiny version **ARM 7** TDMI with **Von-Neumann**-memory interface, **no cache, no MPU or MMU**, simple **integer-ALU**, **short** 3 stage **pipeline**
- Larger versions ARM 9 / 11 / Cortex M / Cortex A with Harvard memory interface, cache, MPU or MMU in different versions with integer and floating-point ALUs, longer pipelines with 5 to 8 stages, DSP- and Java extensions
- **Advanced Risk Machine (ARM)** is a **design house**, which develops the CPU architecture and gives licenses to semiconductor manufacturers. ARM also provides development tools (compiler, debugger). The semiconductor supplier adds proprietary peripherals and memory, i.e. ARM CPUs of different suppliers are not compatible, even though they have the same programming model.
- **Increasing market share**, due to:

  High computing performance/Watt, small chip size, reasonable license fees

Register Set



- **Register set** with 16 registers, (incl. 3 special purpose registers (PC, LR, SP)) and a status register (CPSR), 64bit ARM extends register set to 31 regs R0 - R30
- All Register (incl. PC and SP) can be used as operands in instructions.
- Registers SP and LR will be switched on interrupts (exceptions). After returning from the ISR, the CPU will switch back to the original registers SP and LR again.

Typical RISC Load- and Store Architecture with 3 Address Instructions

- Originally only ~ 40 instructions, all ARM instructions 32bit → *simple instruction decoder*
- Arithmetic-logic instructions use registers or constants only, no memory operands.
- Each arithmetic-logic instruction has 2 operand and 1 result register
  Example: **`ADD R0, R1, R2`** **`R0 = R1 + R2`** *status bits not modified*
- For each instruction the programmer can define, if the status bits (Negative, Zero, Carry, …) shall be changed or not.
  Example: **`ADDS R0, R1, R2`** **`R0 = R1 + R2`** *status bits modified*

5.4 ARM

- For each instruction a condition can be defined. The instruction will only be executed, it the condition is true, otherwise it is executed as a NOP → *in many cases conditional branches (and their problems in pipelined CPUs) can be avoided*

  Example: **`ADDMI  R0, R1, #4   R0 = R1 + 4`** *only executed if the negative Bit (MI=Minus) was set before*

  `ADDMIS R0, R1, #4` *as above, additionally modify the status register*

  Available conditions:

  EQ/NE Equal/Not Equal, **CS/CC** Carry Set/Clear, **MI/PL** Minus/Positive or zero, **VS/VC** Overflow Set/Clear, **GE/GT/LE/LT** Greater Equal/Greater/Less Equal/Less (for signed (2s-complement) numbers), **HI/LS** Higher/Lower or Same (for unsigned numbers)

- The second operand can be shifted to the left or to the right, i.e. multiplied or divided by 2^x, before the instruction is executed. The number of shifts can be specified via a constant or via another register.

  Example: **`ADD R0, R1, R2 ASL #4`**   **`R0 = R1 + (R2<<4) = R1 + R2 ·`** 2^4

  `ADD R0, R1, R2 ASR R3`   **`R0 = R1 + (R2>>R3)= R1 + R2 /`** 2^{R3}

  with **ASL/ASR** Arithmetic Shift Left/Right (for signed numbers), **LSL/LSR** Logic Shift left/right (for unsigned numbers), ROR Rotate to Right

- Modifying the status bits, bit shifting and conditions are also possible with data transport instructions

  Exp.: **`MOVEQS R0, R1 ASL #4`**   **`R0 = R1<<4`**, if Z bit was set, modify status register

Addressing Modes to Load and Store Registers from/to Memory

• **`LDR  Rx, <SourceAddress>`**	Load a register from memory	**`Rx, x=0 … 15`**
• **`STR  Rx, <DestAddress>`**	Store a register to memory	**`Rx, x=0 … 15`**

Conditions for instructions see above.

Addressing modes for memory source and destination **`<…address>`**:

Direct address

Register-indirect address, e.g.	**`[R0]`**	address in R0
	`[R0,#4]`	address is R0 + 4
	`[R0,R1]`	address is R0 + R1
	`[R0,R1,ASL #2]`	address is R0 + (R1<<2)

Pre-Indexed:
In all previous examples the content of R0 and R1 will not change. Adding a **`!`** (Register Writeback) to the braces, e.g. **`[R0, R1]!`**, **`R0`** will be changed to the newly computed address, i.e. here **`R0=R0+R1`**.

Post-Indexed:
With **`[ ]`** (without !) around the first register only, e.g. **`[R0],R1,ASL #2`**, only the first register's content will be used as address, here **`R0`**, but the address register R0 nevertheless will be changed to **`R0=R0+(R1<<2)`**.

- **`LDM`** and **`STM`** can load and store multiple registers with a single instruction.

Subroutine Calls, Link Register LR and Stack

- ARM-CPUs may operate without stack, if subroutines are not nested. A subroutine call `BL subroutine` (**BL** = Branch and Link) stores the return address in register **R14=LR**. Returning from a subroutine uses **MOV PC,LR**, i.e. copying the return address from **LR** to the instruction pointer **PC**. The CPU does not have an explicit Return from Subroutine instruction.
- When subroutine calls are nested, **R14=LR** is overwritten by the second return address. Therefore, the first return address must be manually saved and restored by the programmer, i.e. a push/pop must be simulated.
- Instructions

`STR R0,[SP, #-4]!`	Decrement **SP** and store **R0** on the stack
`LDR R0,[SP],#4`	Read **R0** from stack and increment **SP**

  simulate the well-known PUSH/POP/PULL stack operations of other microprocessors.
  With

  ```
  STMFD SP!, {R0-R4, R6}
  LDMFD SP!, {R0-R4, R6}
  ```

  A list `{…}` of registers can be stored/loaded to/from the stack in a single instruction (including **LR** or **PC**).
- For one-way branches use `B address` (which actually is implemented by `MOV PC,…`).
- Like all other instructions, branch instructions may have a condition.

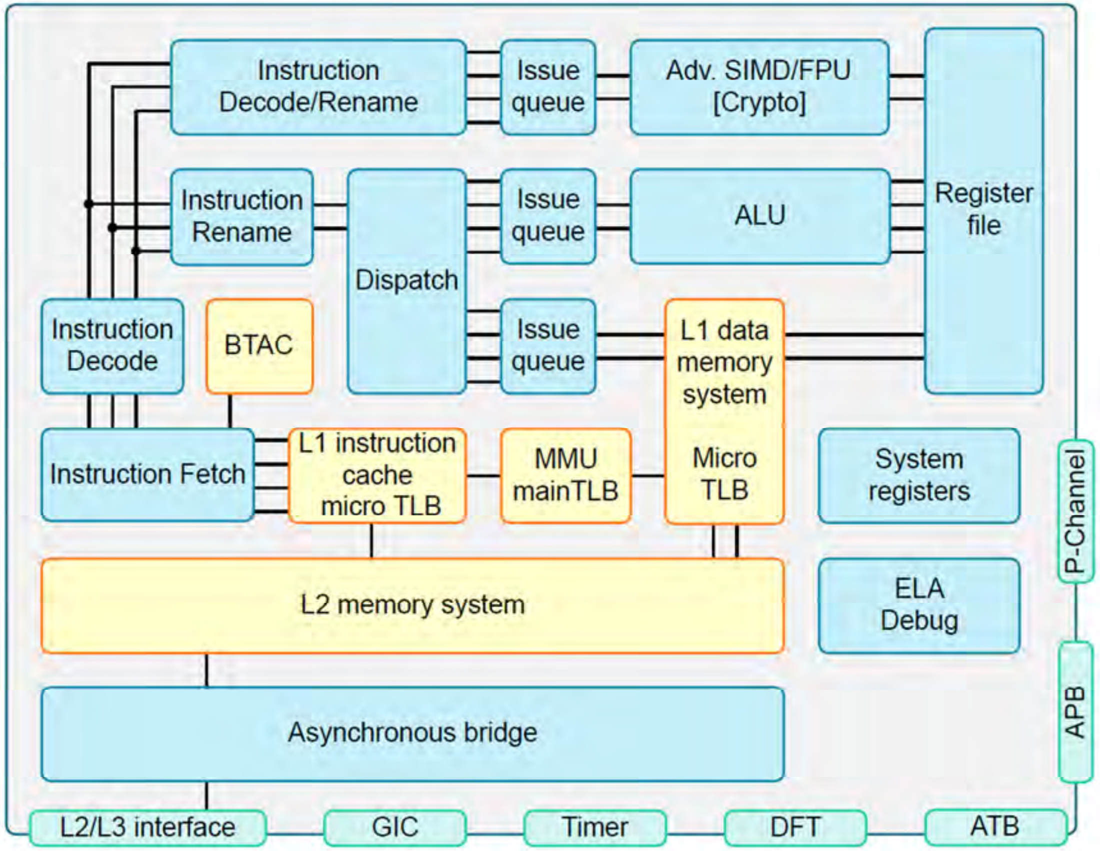
Protection Methods

- Like 80x86-CPUs ARM-CPUs have privileged and non-privileged operating modes (ARM *User Mode*, *System Mode* and various *Exception Modes*).
- Switching modes is done by setting bits in the status register or automatically by an interrupt or exception. With each mode change register CSPR will be saved and registers LR and SP will be switched, i.e. each mode uses its own stack. Mode switching is a priviledged instruction.
- Memory Protection MPU and virtual Memory Management MMU are optional.

Advantages/disadvantages of 32bit RISC-instructions vs. 16bit microcontrollers

- 32bit RISC instructions (*ARM instruction set*) lead to faster, but longer program code compared to typical 8/16bit CISC microcontrollers.
- To reduce the memory size, ARM-CPUs additionally have a set of 16bit instructions (*Thumb instruction set*). Thumb code is approx. 30% shorter, but 40% slower than 32bit code.
- Switching between ARM and Thumb instructions is possible when calling subroutines (*ARM-Thumb-Interworking*)
- Thumb instructions have several restrictions and are much closer to typical CISC microprocessors:
  - Only 2 rather than 3 register operands per instruction (2-address instruction)
  - No conditional execution of instructions, code has to use conditional branches
- The **ARM Cortex M**-family of CPUs, which addresses the classical embedded microcontroller market, only supports a Thumb(2) instruction set (no classical 32bit ARM instructions).

ARM Cortex A-Family: ARM-64bit-Architecture



Cortex A75 (Source: ARM technical reference manual)

- Target: Server market, **attack 80x86**
  → high memory requirements

  → **Cortex Ax** series
- Instruction set **AArch64** with **new 32bit** (!) **opcodes**, **64bit addresses** and **new register model** with 31 general-purpose 64bit CPU registers
- AArch64 instruction set not backwards compatible (new opcodes, modified instruction and register set)
- Additional **AArch32** mode = classical ARM-32bit-architecture, mode switch via software interrupt for backwards compatibility (aka Intel: Now ARM also suffers from its own history!)
- User programs use 32bit or 64bit mode, operating systems must run in 64bit mode
- Multi-Core CPUs with superscalar architecture and up to 3 cache levels in internal Harvard architecture

Appendix
C/C++ Variables and HCS12 Addressing Modes

Sample C and HCS12 assembler code for this appendix can be found in Codewarrior project **AsmIntro3.mcp**.

HCS12 CPU Instructions e.g. **MOVB var1, var2**

Instruction Operands (max. 2)

How does the CPU find the operands? → Addressing Modes

Basic CPU Addressing Modes

1. Register]
2. Constants (immediate)] HCS12 and other CPUs use variants and
3.1 Memory direct] combinations of these addressing modes
3.2 Memory indirect (“pointer”)]

1. HCS12 Register Addressing

implicit **CLRA**
explicit **TFR D, X**

2. Constants, HCS12 Immediate Addressing **LDAB #34**

Constants are marked by a **hash sign #**. The constant is part of the instruction, i.e. requires space in code memory, but no space in data memory. You can use an ASCII character instead of a number, e.g. #‘A’. The compiler will convert it into the respective ASCII code (= number). Constants in C/C++ need not be marked **varB = 34**

Memory Organization

Example in C

```
char var1      = 0x12;

int  var2      = 0x3456;


char *p        = &var1;


char var3{2]   = { 0x70,
                   0x71 };
int  var4{2]   = { 0x8008,



                   0x8118 };
char var5      = 0x23;
```

HCS12 Memory

Address (16 bit)	**Value** (memory contents 8 bit)	
0x1000	0x12	var1
0x1001	MSB 0x34	var2
0x1002	0x56 LSB	
0x1003	0x10	P
0x1004	0x00	
0x1005	0x70	var3[0]
0x1006	0x71	var3[1]
0x1007	0x80	“var4[0]”
0x1008	0x08	
0x1009	0x81	“var4[1]”
0x100A	0x18	
0x100B	0x23	var5

Features of Variables, Pointers and Arrays

		Example
• **Scalar** variables	have a value	**var2 = 0x3435**
	a size in memory	2 byte (1, 2 or 4 byte depends on data type)
	an address in C HCS12	**&var2 = 0x1001** **#var2 = 0x1001** address is the address of the first byte in big endian sequence
• **Pointer** variables	have a value	**p = 0x1000**
pointer variable = variable which’s value is an address	a size in memory	2 byte (always, data type does not matter!)
	an address C HCS12	**&p = 0x1003** **#p = 0x1003**
	point to another variable	***p = *0x1000 = 0x12** value of a pointer variable is the address of the other variable

Note: “Pointer arithmetic” allows access across variable borders

p + 11 = 0x1000 + 0x0B = &var5

&var1 + 0x0B = 0x100B = &var5

- **Arrays**

have elements with values
(like variables)
	`var3[0] = 0x70`
	`var3[1] = 0x71`

C arrays are indexed per element (element size does not matter)	`var4[1] = 0x8118`

HCS12 arrays are indexed per memory byte
(element size is important)
	`var4,X = 0x81 if X = 2`
	`var4,X = 0x18 if X = 3`
	HCS12 index = C index · element size

Array elements have addresses	`&var3[1] = 0x1006`
The array name can be used instead of the address of its first element	`var3 = &var3[0] = 0x1005` I prefer &var3 instead of var3, but in ANSI C strict this syntax is an error. The array name is not a pointer variable, but a constant address!

Often a pointer variable is used to access array elements. The pointer may be indexed.	`p = &var3[0] = 0x1005`
	`*p = *0x1005 = 0x70`
	`*(p+1) = p[1] = 0x71`

3.1 HCS12 Direct Memory Addressing

LDAB var1

in C/C++: **varB = var1**

The instructions read the **VALUE** of variable var1 (not the address).

When the variable is declared, the compiler will reserve space for the variable in memory.

When the variable is used, the 16bit **ADDRESS** (of the first byte) of the variable will be part of the instruction. Addresses are fixed at compile time and don’t change at runtime!

The CPU has no idea of data types and does not respect borders between variables.

How does the CPU know, how many bytes to read or write?

- Either the instruction specifies it … **MOVW p, var1**

  copies 2 bytes from p into var1 and the first byte of var2

- … or the CPU uses the size of the register operand **LDD var1**

  copies 2 bytes (var1 and the first byte of var2) into register D

NOTE:

LDX #$100B	X = 0x100B	Numbers marked with # are constants
LDAB $100B	B = *0x100B = 0x23	Numbers without # instead of a variable name are direct addresses
LDAB var5	B = var5 = 0x23	If we want the value of a variable, we use the name of the variable
LDX #var5	X = &var5 = 0x100B	If we want the address of a variable, we mark it

3.2 Indirect Addressing = “Pointers”

Indirect addressing is needed, when the operand address is variable and thus must be calculated by the CPU during runtime and not by the compiler during compile time. The HCS12 supports a number of variants of indirect addressing:

- **HCS12 Register-indirect Addressing with Index or Offset** (single indirection)
  - Read (write) an **array element with** a variable **index** **LDAB var3, X**

    name of the array index in X or Y

    in C/C++: **varB = var3[X]** =*(&var3+X)

    The effective address of the array operand (=”pointer” to the operand) is var3 + X = &var3[0] + X

    Note: If the array index is constant, rather than loading the constant into X or Y use direct addressing, which is faster **LDAB var3+1**
  - Read (write) a variable in memory **with** a **pointer** in register X or Y **and** an **offset**

    assume X = &var3[0] **LDAB 1, X**

    in C/C++: **varB = *(X+1)** =*(&var3+1)

    The result is exactly the same as LDAB var3, X above when X=1

    If the offset is variable rather than constant, use **LDAB D, X**

    in C/C++: **varB = *(X+D)**

- **HCS12 Register-indirect Addressing with Pre/Post-Inkrement/Decrement**

  Read (write) a variable in memory with a pointer in register X or Y and increment or decrement the pointer before or after using it **LDAB 3, X+**

  X+ post-increment, +X pre-increment, X– post-decrement, –X pre-decrement

  in C/C++: **varB = *X**

  X = X + 3

- **HCS12 Memory-indirect Addressing with Index / Offset** (single indirection)

  Read (write) a variable in memory using two pointers. The first pointer is in register X or Y and points to a second pointer in memory. The pointer in memory then points to the variable:

  ```
  LDX     #$1003
  LDAB    [ 0, X ]
  ```

  in C/C++: **varB = *(*(X+0))** = 0x12

  Assuming X=0x1003, the **first pointer 0, X** points to memory address 0x1003. The value at this address is used as a **second pointer [...]**, pointing to memory address 0x1000, from where the CPU finally fetches the **data** 0x12 to load into register B.

  Note: Memory-indirect can be avoided by using register-indirect twice, e.g. LDAB [0,X] →

  ```
  LDX   0,X
  LDAB 0,X
  ```

The constant offset/index refers to X, i.e. to the first pointer. The pointer in memory can be interpreted as an array of pointers, i.e.

	Address (16 bit)	**Value** (memory contents 8 bit)	
`char var1      = 0x12;`	0x1000	0x12	**var1**
. . .	...	...	...
`char var5      = 0x23;`	0x100B	0x23	**var5**
`char *pP{2]    = { &var1,`	0x100C	0x10	**“pP[0]”**
	0x100D	0x00	
`&var5 };`	0x100E	0x10	**“pP[1]”**
	0x100F	0x0B	

Example:

```
LDX    #pP
LDAB [2, X]
```

X = &pP[0] = 0x100C

B = *(*(X+2))

= *(*0x100E) = *0x100B = 0x23

In C/C++: **varB = *pP[1]**

Note: C index is element-wise, while HCS12 index is byte-wise

Again memory-indirect can be substituted by

```
LDX   2, X
LDAB 0, X
```

Numbers and Character Coding

- The computer codes **integer numbers** in a binary format with a length of 8, 16, 32, … bit, in (HCS12) C **`char`**, **`int`**, **`long`**, …
- For **real numbers** 32bit or 64 bit IEEE 754 format is used (in C: **`float`**, **`double`**).
- To store **characters** computers typically use 8 bit **ASCII** coding, in C **`char`**. Note: **`char`** is also used for 8 bit integer numbers.
- **Text** with more than one character is stored in C in **`char arrays`** in ASCIIZ format, i.e. ASCII coding plus a 0 byte to mark the end of the text.
- C++ and other programming languages use a special format called **`string`**, which is a wrapper for these arrays, which simplifies the handling of non-ASCII character sets like 16 bit Unicode or variable length UTF-8 text.

For the CPU numbers or text is a sequence of bits with known length which can be handled via the CPU's binary operations. To interface with humans, these bit sequences need to be converted to and from human readable. In C the conversion for variables is handled via the format strings of input/output functions like **`printf()`**, **`scanf()`** or explicit conversion functions like **`atoi()`**, **`itoa()`**, … or via the compiler (for constants). Keyboard input or display/printer output always uses TEXT format, i.e. ASCII characters, even it it looks like a number!